Computer Science

Memory Subsystem Performance of Programs with Intensive Heap Allocation

Amer Diwan David Tarditi Eliot Moss [1]

December 13, 1993

CMU-CS-93-227

Carnegie Mellon

1

Memory Subsystem Performance of Programs with Intensive Heap Allocation

Amer Diwan David Tarditi Eliot Moss [1]

December 13, 1993

CMU-CS-93-227

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted for publication. This paper is also published as Fox Memorandum CMU-CS-FOX-93-07. A paper containing some of the results presented in this paper will appear in *21st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Portland, Oregon, January 16-19, 1994.

[1] Authors' addresses: Amer Diwan and Eliot Moss, Department of Computer Science, University of Massachusetts, Amherst, MA 01003-4610. e-mail: diwan@cs.umass.edu, moss@cs.umass.edu. David Tarditi, Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA, 15213-3891. e-mail: dtarditi@cs.cmu.edu.

This research is sponsored by the Defense Advanced Research Projects Agency, DoD, through ARPA Order 8313, and monitored by ESD/AVS under contract F19628-91-C-0168. David Tarditi is also supported by an AT&T PhD Scholarship. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the United States Government, or AT&T.

Abstract

Heap allocation with copying garbage collection is a general storage management technique for modern programming languages. It is believed to have poor memory subsystem performance. To investigate this, we conducted an in-depth study of the memory subsystem performance of heap allocation for memory subsystems found on many machines. We studied the performance of mostly-functional Standard ML programs which made heavy use of heap allocation. We found that most machines support heap allocation poorly. However, with the appropriate memory subsystem organization, heap allocation can have good performance. The memory subsystem property crucial for achieving good performance was the ability to allocate and initialize a new object into the cache without a penalty. This can be achieved by having subblock placement with a subblock size of one word with a write allocate policy, along with fast page-mode writes or a write buffer. For caches with subblock placement, the data cache overhead was under 9% for a 64K of larger data cache; without subblock placement the overhead was often higher than 50%.

1 Introduction

Heap allocation with copying garbage collection is widely believed to have poor memory subsystem performance [30, 38, 48, 49, 50]. To investigate this, we conducted an extensive study of memory subsystem performance of heap allocation intensive programs on memory subsystem organizations typical of many workstations. The programs, compiled with the SML/NJ compiler [4], do tremendous amounts of heap allocation, allocating one word every 4 to 10 instructions. The programs used a generational copying garbage collector to manage their heaps. To our surprise, we found that for some configurations corresponding to actual machines, such as the DECStation 5000/200, the memory subsystem performance was comparable to that of C and Fortran programs [12]: programs ran only 3 to 13% slower due to data cache misses than they would have with an infinitely fast memory. For other configurations, the slowdown due to data cache misses was often higher than 50%.

The memory subsystem features important for achieving good performance with heap allocation are subblock placement with a subblock size of one word, combined with write-allocate on write-miss, page-mode writes, and cache sizes of 32K or larger. Heap allocation performs poorly on machines whose caches are smaller than the allocation area of the programs (256K or larger for the benchmarks studied here) and do not have one or more of the features mentioned above; this includes most current workstations.

Our work differs from previous reported work [30, 38, 48, 49, 50] on memory subsystem performance of heap allocation in two important ways. First, previous work used the *overall miss ratio* as the performance metric, which is a misleading indicator of performance. The overall miss ratio neglects the fact that read and write misses may have different costs. Also, the overall miss ratio does not reflect the rates of reads and writes, which may substantially affect performance. We use memory subsystem contribution to cycles per instruction (CPI) as our performance metric, which accurately reflects the effect of the memory subsystem on program running time. Second, previous work did not model the entire memory subsystem: it concentrated solely on caches. Memory subsystem features such as write buffers and page-mode writes interact with the costs of hits and misses in the cache and should be simulated to give a correct picture of memory subsystem behavior. We simulate the entire memory subsystem.

We did the study by instrumenting programs to produce traces of all memory references. We fed the references into a memory subsystem simulator which calculated a performance penalty due to the memory subsystem. We fixed the architecture to be the MIPS R3000 [28] and varied cache configurations to cover the design space typical of workstations such as DECStations, SPARCStations, and HP 9000 series 700. We studied eight substantial programs.

We varied the following memory subsystem parameters: cache size (8K to 512K), cache block size (16 or 32 bytes), write miss policy (write allocate or write no-allocate), subblock placement (with and without), associativity (one and two way), TLB sizes (1 to 64 entries), write buffer depth (1 to 6 deep), and page-mode writes (with and without). We simulated only split instruction and data caches, *i.e.*, no unified caches. We report data only for write-through caches but the results extend easily to write-back caches.

Section 2 gives background information. Section 3 describes related work. Section 4 describes the simulation methods, the benchmarks, and the metrics used to measure memory subsystem performance. Section 5 presents the results of the simulation studies, an analysis of those results, validation of those results, and an analytical model which is used to extend the results to programs with different allocation behavior. Section 6 suggests promising areas for future work. Section 7 concludes.

2 Background

The following sections describe memory subsystems, copying garbage collection, SML, and the SML/NJ compiler.

2.1 Memory subsystems

This section reviews the organization of memory subsystems. Terminology for memory subsystems is not standardized; we use Przybylski's terminology [39].

It is well known that CPUs are getting faster relative to DRAM memory chips [37]: main memory cannot supply the CPU with instructions and data fast enough. A solution to this problem is to use a cache, a small fast memory placed between the CPU and main memory that holds a small subset of memory. If the CPU reads a memory location which is in the cache, the value is returned quickly. Otherwise the CPU must wait for the value to be fetched from main memory.

Caches work by reducing the average memory access time. This is possible since memory accesses exhibit *spatial* and *temporal* locality. Temporal locality means that a memory location that was referenced recently will probably be referenced again soon and is thus worth storing in the cache. Spatial locality means that a memory location near one which was referenced recently will probably be referenced soon. Thus, it is worth moving the neighboring locations to the cache.

2.1.1 Memory subsystem organization

This section describes cache organization for a single level of caching. A cache is divided into *blocks* each of which has an associated *tag*. A cache block represents a block of memory. The tag for a cache block indicates what memory block it holds. Cache blocks are grouped into *sets*. A memory block may reside in the cache in exactly one set, but may reside in any block within the set. A cache with sets of size n is said to be *n-way associative*. If $n=1$, the cache is called *direct-mapped*. Some caches have valid bits, to indicate what sections of a block hold valid data. A *subblock* is the smallest part of a cache with which a valid bit is associated. In this paper, *subblock placement* implies a subblock of one word, *i.e.*, valid bits are associated with each word. Moreover, on a read miss, the whole block is brought into the cache not just the subblock that missed. Przybylski [39] notes that this is a good choice.

A memory access to a location which is resident in the cache is called a *hit*. Otherwise, the memory access is a *miss*.

A read request for memory location m causes m to be mapped to a set. All the tags and valid bits (if any) in the set are checked to see if any block contains m. If a cache block contains m, the word corresponding to m is selected from the cache block. A read miss is handled by copying the missing block from the main memory to the cache.

The way write requests are handled depends upon the *write policy*. The write policy describes whether writes to the cache go immediately to main memory. In a *write-through* cache, writes to the cache immediately go to main memory. In a *write-back* cache, writes to the cache do not immediately go to main memory; they are just written to the cache. The writes eventually go to main memory when a memory block is removed from the cache. Write-back caches use less bus bandwidth than write-through caches, because multiple writes to the same location may be coalesced into one write to main memory by the write back cache, whereas all the writes would go to main memory with a write through cache. See [27] for a discussion of the relative merits of write back and write through caches.

A write hit is always written to the cache. There are several policies for handling a write miss, which differ in their performance penalties. For each of the policies, the actions taken on a write miss are:

1. write-no-allocate:

   - Do not allocate a block in the cache

- Send the write to main memory, without putting the write in the cache.

2. write-allocate, no-subblock placement:

   - Allocate a block in the cache.
   - Fetch the corresponding memory block from main memory.
   - Write the word to the cache (and to memory if write through).

3. write-allocate, subblock placement[1]:
   If the tag matches but the valid bit is off:

   - Write the word to the cache (and to memory if write through).

   If the tag does not match:

   - Allocate a block in the cache.
   - Write the word to the cache (and to memory if write through).
   - Invalidate the remaining words in the block.

Write allocate/subblock placement will have a lower write miss penalty than *write allocate/no subblock placement* since it avoids fetching a memory block from main memory. In addition, it will have a lower penalty than *write no allocate* if the written word is read before being evicted from the cache. See Jouppi [27] for more information on write-miss policies.

A miss is a *compulsory miss* if it is due to a memory block being accessed for the first time. A miss is a *capacity miss* if it results from the cache not being large enough to hold all the memory blocks used by a program. The capacity misses for a given cache size correspond to the misses in a fully associative cache of the same size with an LRU replacement policy minus the compulsory misses. It is a *conflict miss* if it results from two memory blocks mapping to the same set. [25]

The memory subsystem bandwidth may be increased by using separate caches for instructions and data. This is called a *split instruction-data cache*. The memory bandwidth is increased since a data access and an instruction fetch may be handled at the same time. A cache where instructions and data gc to the same cache is called a *unified* cache. This paper presents results only for split instruction-data caches.

A *write buffer* may be used to reduce the cost of writes to main memory. A *write buffer* is a queue containing writes that are to be sent to main memory. When the CPU does a write, the write is placed in the write buffer and the CPU continues without waiting for the write to finish. The write buffer retires entries to main memory using free memory cycles. There are situations when the write buffer is not fully effective in preventing stalls on writes to main memory. First, if the CPU writes to a full write buffer, the CPU must wait for an entry to become available in the write buffer. Second, if the CPU reads a location which is queued up in the write buffer, the CPU may need to wait until the write buffer is empty. Third, if the CPU issues a read to main memory while a write is in progress, the CPU must wait for the write to finish.

Main memory is divided into DRAM pages. *Page-mode writes* reduce the latency of writes to the same DRAM page when there are no intervening memory accesses to another DRAM page. Page-mode writes work as follows. DRAMs are organized internally as arrays, and all the locations on a DRAM page reside on the same row in the DRAMs which implement main memory. This fact can be used to speed up a sequence of writes to one DRAM page. A DRAM is updated in a read-modify-write cycle: an array row is latched into a row buffer, the row buffer is modified, and then written back to the array. A sequence of writes to the same DRAM page can update the row while it is held in the row buffer, and avoid the read and write cycles for all but the first and last writes, respectively. This improves write speed significantly. For example, on a DECStation 5000/200, a non-page-mode write takes 5 cycles, while a page-mode write takes 1 cycle. Main memory is said

[1] Recall subblock size is assumed to be 1 word.

```
% check for heap overflow
cmp alloc+12,top
branch-if-gt call-gc
% write the object
store tag,(alloc)
store ra,4(alloc)
store rd,8(alloc)
% save pointer to object
move alloc+4,result
% add 12 to alloc pointer
add alloc,12
```

Figure 1: Pseudo-assembly code for allocating an object

to be operating in page mode when DRAM rows are held in row buffers across memory accesses. It is thrown out of page mode when a memory access to a different DRAM page is made. It may also be thrown out of page mode for other machine-specific reasons (such as refreshes). Page-mode writes are especially effective at handling writes with high spatial locality, such as those seen when saving registers at a procedure call or when doing sequential allocation.

2.1.2 Memory subsystem performance

This section describes two metrics for measuring the performance of memory subsystems. One popular metric is the *cache miss ratio*. The cache miss ratio is the number of memory accesses which miss divided by the total number of memory accesses. Since different kinds of memory accesses usually have different miss costs, it is useful to have miss ratios for each kind of access.

Cache miss ratios alone do not measure the impact of the memory subsystem on overall system performance. A metric which better measures this is the contribution of the memory subsystem to CPI (cycles per useful instruction[2]). CPI is calculated for a program as *number of CPU cycles to complete the program / total number of useful instructions executed.* It measures how efficiently the CPU is being utilized. The contribution of the memory subsystem to CPI is calculated as *number of CPU cycles spent waiting for the memory subsystem / total number of useful instructions executed.* As an example, on a DECStation 5000/200, the lowest CPI possible is 1, completing one instruction per cycle. If the CPI for a program is 1.50, and the memory contribution to CPI is 0.3, 20% (0.3/1.5) of the CPU cycles are spent waiting for the memory subsystem (the rest may be due to other causes such as nops, multi-cycle instructions like integer division, etc.). CPI is machine dependent since it is calculated using actual penalties.

2.2 Copying garbage collection

A copying garbage collector [22, 14] reclaims an area of memory by copying all the live (non-garbage) data to another area of memory. This means that all data in the garbage-collected area is now garbage, and the area can be re-used. Since memory is always reclaimed in large contiguous areas, objects can be sequentially allocated from such areas at the cost of only a few instructions. Figure 1 gives an example of pseudo-assembly code for allocating a **cons** cell. **ra** contains the **car** cell contents, **rd** contains the **cdr** cell contents, **alloc** is the address of the next free word in the allocation area, and **top** contains the end of the allocation area.

[2] All instructions besides nops are considered as useful. A nop (null operation) instruction is a software-controlled pipeline stall.

The SML/NJ compiler uses a simple generational copying garbage collector [2]. Memory is divided into an old generation and an allocation area. New objects are created in the allocation area; garbage collection copies the live objects in the allocation area to the old generation freeing up the allocation area. Generational garbage collection relies on the fact that most allocated objects die young; thus most objects (about 99% [4, p. 206]) are not copied from the allocation area. This makes the garbage collector efficient, since it works mostly on an area of memory where it is very effective at reclaiming space.

The most important property of a copying collector with respect to memory subsystem behavior is that allocation initializes memory which has not been touched in a long time and is thus unlikely to be in the cache. This is especially true if the allocation area is large relative to the size of the cache since allocation will knock everything out of the cache. This means that caches which cannot hold the allocation area will incur a large number of write misses.

For example consider the code in Figure 1. Assume that a cache write miss costs 16 CPU cycles and that the block size is 4 words. On average, every fourth word allocated causes a write miss. Thus, the average memory subsystem cost of allocating a word on the heap is 4 cycles. The average cost for allocating a cons cell is seven cycles (at one cycle per instruction) plus 12 cycles for the memory subsystem overhead. Thus, while allocation is cheap in terms of instruction counts, it may be expensive in terms of machine cycle counts.

2.3 Standard ML

Standard ML (SML) [35] is a call-by-value, lexically scoped language with higher-order functions, with many of the features deemed good by the programming language community. It has garbage collection to automate the management of heap storage. This eliminates two common kinds of programming errors that occur with explicit storage management, memory leaks and dangling pointers. Memory leaks occur when memory is never deallocated, and dangling pointers occur when memory is deallocated too soon. SML is statically typed, so many programming errors are caught at compile-time. The type system is polymorphic, and types are inferred automatically by the compiler, so the type system is flexible yet not an impediment to the programmer. The language is provably safe, that is, there are no holes in the type system and a program always has a well-defined behavior. SML has a sophisticated module system to support the development of large programs. The module system provides for static type-checking of the interfaces between modules, as in Ada and Modula-3. It has a dynamically-scoped exception mechanism to allow programs to handle unusual conditions.

SML encourages a non-imperative programming style. Variables cannot be altered once they are bound, and by default data structures cannot be altered once they are created. Lisp's `rplaca` and `rplacd` do not exist for the default definition of lists in SML. The only kinds of assignable data structures are **ref** cells and arrays[3], which must be explicitly declared. To emphasize the point, assignments are permitted but discouraged as a general programming style. The implications of this non-imperative programming style for compilation are clear: SML programs tend to do more allocation and copying than programs written in imperative languages.

SML is most closely related to Lisp and Scheme [41]. Implementation techniques for one of these languages are mostly applicable to the other languages, with the following caveats: SML programs tend to be less imperative than Lisp or Scheme programs and Scheme and SML programs use functions calls more frequently than Lisp, since recursion is the usual way to achieve iteration in those languages.

2.4 SML/NJ compiler

The SML/NJ compiler [4] is a publicly available compiler for SML. We used version 0.91. The compiler concentrates on making allocation cheap and function calls fast. Allocation is done in-

[3] Although the language definition omitted arrays, all implementations have arrays.

line, except for the allocation of arrays. Aggressive function inlining is used to eliminate functions calls and their associated overhead. Function arguments are passed in registers when possible. and register targeting is used to minimize register shuffling at function calls. A split caller/callee-save register convention is used to avoid excessive spilling of registers [8]. The compiler also does constant-folding, limited code hoisting. uncurrying, and instruction scheduling.

The most controversial design decision in the compiler was to allocate procedure activation records on the heap instead of the stack [1. 6]. In principle. the presence of higher-order functions means that procedure activation records must be allocated on the heap. With a suitable analysis. a stack can be used to store most activation records [31]. However. using only a heap simplifies the compiler. the run-time system [3]. and the implementation of first-class continuations [23]. The decision to use only a heap was controversial because it greatly increases the amount of heap allocation. which is believed to cause poor memory subsystem performance.

3 Related Work

There have been many studies of the cache behavior of systems using heap allocation and some form of copying garbage collection. Peng and Sohi [38] examined the data cache behavior of small Lisp programs. They used trace-driven simulation. and proposed an ALLOCATE instruction for improving cache behavior. which allocates a block in the cache without fetching it from memory. Wilson *et al.* [48. 49] argued that cache performance of programs with generational garbage collection will improve substantially when the youngest generation fits in the cache. Koopman *et al.* [30] studied the effect of cache organization on combinator graph reduction. an implementation technique for lazy functional programming languages. They observed the importance of a write-allocate policy with subblock placement for improving heap allocation. Zorn [50] studied the impact of cache behavior on the performance of a Common Lisp system. when stop-and-copy and mark-and-sweep garbage collection algorithms were used. He concluded that when programs are run with mark-and-sweep they have substantially better cache locality than when run with stop-and-copy.

Our work differs from previous work in two important ways. First. previous work used the overall miss ratio as the performance metric. which is a misleading indicator of performance. The overall miss ratio neglects the fact that read and write misses may have different costs. Also. the overall miss ratio does not reflect the rates of reads and writes. which may substantially affect performance. We use memory subsystem contribution to CPI as our performance metric. which accurately reflects the effect of the memory subsystem on program running time. Second. previous work did not model the entire memory subsystem: it concentrated solely on caches. Memory subsystem features such as write buffers and page-mode writes interact with the costs of hits and misses in the cache and should be simulated to give a correct picture of memory subsystem behavior. We simulate the entire memory subsystem.

Appel [4] estimated CPI for the SML/NJ system on a single machine using elapsed time and instruction counts. His CPI differs substantially from ours. Apparently instructions were under-counted in his measurements [5].

Jouppi [27] studied the effect of cache write policies on the performance of C and Fortran programs. Our class of programs is different from his. but his conclusions support ours: that a write-allocate policy with subblock placement is a desirable architecture feature. He found that the write miss ratio for the programs he studied was comparable to the read miss ratio. and that write-allocate with subblock placement eliminated many of the write misses. For programs compiled with the SML/NJ compiler. this is even more important due to the high number of write misses caused by allocation.

4 Methodology

We used trace driven simulations to evaluate the memory subsystem performance of programs compiled with the SML/NJ compiler. For trace driven simulations to be useful, there must be an accurate simulation model and a good selection of benchmarks. Simulations that make simplifying assumptions about important aspects of the system being modeled can yield misleading results. Toy benchmarks, or benchmarks that are not representative of the kinds of tasks the system is normally used for, can be equally misleading. In this work, much effort has been devoted to addressing these issues.

Section 4.1 describes our trace generation and simulation tools. Section 4.2 states our assumptions and argues that they are reasonable. Section 4.3 describes and characterizes the benchmark programs used in this study. Section 4.4 describes the metrics used to present memory subsystem performance.

4.1 Tools

We extended QPT (Quick Program Profiler and Tracer) [33, 9, 32] to produce memory traces for SML/NJ programs. QPT rewrites an executable program to produce compressed trace information; QPT also produces a program specific regeneration program that expands the compressed trace into a full trace. Because QPT operates on the executable program, it can trace both the SML code and the garbage collector (which is written in C). The significant trace compression achieved by QPT allowed us to send traces to faster machines where they could be regenerated and simulated quickly: about 50 μs to regenerate and simulate each memory reference on an HP 9000 model 720 machine[4].

Code produced by the SML/NJ compiler presents three problems for QPT. First, SML/NJ puts its code in the heap. Since SML/NJ uses a copying collector, code can be moved just like data. This creates numerous problems; we solve them by putting SML/NJ code in the text segment, so it is never garbage collected. Second, programs compiled with the SML/NJ compiler have no symbol table information. SML/NJ makes the problem worse by interleaving data with the code. QPT needs a symbol table to find all the code. Third, SML/NJ often implements function calls using indirect jumps. QPT needs to know all the program points that could be targets of an indirect jump. We solved both problems by modifying SML/NJ to produce tables that enable QPT to find all targets of indirect jumps and to separate code from data; we enhanced QPT to use this information.

We used Tycho [24] for the memory subsystem simulations. Tycho uses a special case of *all-associativity simulation* [34] to simulate multiple caches concurrently. We extended Tycho in four important ways. First, we extended Tycho to separate read misses from write misses. Second, we changed Tycho to simulate separate data and instruction caches simultaneously. Third, we added a write buffer simulator to Tycho. The write buffer simulator can concurrently simulate a write buffer for each cache organization being simulated by Tycho. The write buffer simulator also takes page-mode writes and memory refreshes into consideration. Fourth, we added the *write no allocate* write miss policy to Tycho.

We obtained allocation statistics by using an allocation profiler built into SML/NJ. The profiler instruments intermediate code to increment appropriate elements of a *count array* on every allocation. We extended this profiler to count the number of assignments done by SML/NJ programs.

4.2 Simplifications and Assumptions

We wanted to simulate the memory systems as completely as we could. Thus, we tried to minimize assumptions which might reduce the validity of our data. This section describes all the important

[4] While doing cache simulations we were also collecting additional data, such as garbage collection overheads, which slowed down the simulations substantially.

assumptions made in this study and argues that they are reasonable.

1. *Simulating write allocate/subblock placement with write allocate* [illegible] cho does not simulate subblock placement so we approximate [illegible] *cate/no subblock* and ignoring the reads from memory that occur [illegible] cause a small inaccuracy in the CPI numbers. The following example [illegible] when the simplification fails.

   Let us suppose we have a cache block size of 2 words and a subblock [illegible] program issues a write to the first word. Further assume that the write [illegible] placement, the word will be written to the cache and the second [illegible] will be invalidated. However, the simplified model will mark both [illegible] write. If the program subsequently issues a read of the second word [illegible] regarded as a hit. Thus the CPI reported for caches with subblock [illegible] than the actual CPI. This is however a rare occurrence since SML programs [illegible] assignments (see Section 4.3) and most writes are to sequential locations.

2. *Ignoring the effects of context switches and system calls.* Context switches (especially [illegible] caused by system calls) can affect cache performance significantly [36]. We ignore [illegible] it is an operating system issue that affects all programs, not just programs that are [illegible] intensive.

3. *Pessimistic simulation of partial word writes.* Most memory subsystems use a [illegible] smallest addressable unit and also maintain error checking information on a word [illegible] Thus, writes to partial words (bytes, half-words, *etc.*) are more expensive than full word [illegible] since the enclosing word needs to be read, modified, its error checking information, and [illegible] written back. We charge 11 cycles for each partial-word write regardless of whether [illegible] is in the cache. If the word is not in the cache, the cache block is not fetched from [illegible] Also, the write is not queued up in the write buffer. This is mostly consistent with the DECStation 5000/200 model of partial word writes; the key difference is that we are always assuming the worst case scenario (which is probably rare in practice).

   This inaccuracy, however, does not have any significant impact on the accuracy of the simulations; the CPI contribution of partial word writes is negligible even with this pessimistic model (see Section 5).

4. *The simulations are driven by virtual addresses.* The caches in many current machines are physically indexed (notable exceptions are the SPARCs and HP series 700). This can be a problem since the virtual address to physical address mapping can affect the conflicts in the cache. However some virtual to physical mapping schemes (e.g., a variation of *Page Coloring* used in the MIPS operating system) yield similar intra-process cache conflicts as if the cache was virtually indexed [29]. Thus, the simplification is reasonable.

5. *Placing code in the text segment instead of the heap.* This improves performance over the unmodified SML/NJ system by reducing garbage collection overhead, since code is never copied, and by avoiding instruction cache flushes after garbage collections.

6. *Used default compilation settings for SML/NJ.* Default compilation settings enable extensive optimization. Evaluating the impact of these optimizations on cache behavior is beyond the scope of this paper.

7. *Used default garbage collection settings*

   We used the default strategy for sizing the allocation area and the old generation [2]. The heap is sized as **r** times the size of the old generation after the old generation is collected, where **r** is the desired *ratio* of heap size to live data. **r=5** was used for all the program runs. The allocation area is sized as one-half of the free space (the heap space not occupied by the old generation). As the old generation grows after each collection of the allocation area, the free space decreases and the allocation area decreases. This continues until the old generation is collected.

We did not investigate the interaction of the sizing strategy and cache size [49]. When the allocation area is larger than the cache, it may be possible to improve program locality by decreasing the size of the allocation area so that it fits in the cache. However, this would probably increase garbage collection costs. Understanding these tradeoffs is beyond the scope of this paper.

In addition to the ratio, the garbage collector is controlled by the *softmax* and the *initial heap size*. The softmax is a desired upper limit on the heap size which is exceeded only to prevent programs from running out of space. The softmax was 20M: the benchmark programs never reached this limit and were able to always resize their heaps to maintain the desired ratio of 5. The initial heap size was 1M.

8. *MIPS as a prototypical RISC machine.* All the traces are for the DECStation 5000/200, which uses a MIPS R3000 CPU. The results should carry over to other RISC machines but we do not know how applicable the results are to CISC machines.

9. *All instructions take one cycle with a perfect memory subsystem.* On the DECStation5000/200, this is not true for some instructions (such as multiply, etc.). As far as the memory subsystem performance is concerned, multi-cycle instructions change only the write buffer penalties: multi-cycle instructions can give the write buffer more opportunities to retire writes. Section 5.4 shows that the write-buffer overhead is small: thus the inaccuracy introduced by this assumption will be negligible.

10. *Assuming CPU cycle time does not vary with memory organization.* The CPI calculations assume that the CPU cycle time remains the same for different memory organizations. This may not be the case, since the CPU cycle time depends on the cache access time, which may be different for different cache organizations. For example, a 128K cache may take longer to access than an 8K cache.

4.3 Benchmarks

Table 1 describes the benchmark programs[5]. *Knuth-Bendix, Lexgen, Life, Simple, VLIW*, and *YACC* are identical to the benchmarks measured by Appel [4][6]. Table 2 gives the sizes of the benchmarks in terms of lines of SML code (excluding comments and blank lines), maximum heap size in kilobytes, size of the compiled code in kilobytes (does not include the garbage collector and other run-time support code which is about 60K)[7], and run time, in seconds, on a DECStation 5000/200. The run times are the minimum of five runs (see Section 5.6).

Table 3 characterizes the benchmark programs according to the number and kinds of memory references they do. All numbers are reported as a percentage of instructions. The *Reads*, *Writes*, and *Partial writes* columns list the reads, full-word writes, and partial-word writes done by the program *and* the garbage collector; the *assignments* column lists the non-initializing writes done by the program only. The *Nops* column lists the nops executed by the program and the garbage collector. Note that all the benchmarks have long traces: most related works use traces that are an order of magnitude smaller. Also, note that the benchmark programs do few assignments: the majority of the writes are initializing writes.

[5] Available from the authors.

[6] The description of these benchmarks have been copied from [4].

[7] The code size includes 207K for the standard libraries.

Program	Description
CW	The Concurrency Workbench [15] is a tool for analyzing networks of finite state processes expressed in Milner's Calculus of Communicating Systems. The input is the sample session from Section 7.5 of [15].
Knuth-Bendix	An implementation of the Knuth-Bendix completion algorithm, implemented by Gerard Huet, processing some axioms of geometry.
Lexgen	A lexical-analyzer generator, implemented by James S. Mattson and David R. Tarditi [7], processing the lexical description of Standard ML.
Life	The game of Life, written by Chris Reade [40], running 50 generations of a glider gun. It is implemented using lists.
PIA	The Perspective Inversion Algorithm [47] decides the location of an object in a perspec.ive video image.
Simple	A spherical fluid-dynamics program, developed as a "realistic" FORTRAN benchmark [16], translated into ID [21], and then translated into Standard ML by Lal George.
VLIW	A Very-Long-Instruction-Word instruction scheduler written by John Danskin.
YACC	A LALR(1) parser generator, implemented by David R. Tarditi [41], processing the grammar of Standard ML.

Table 1: Benchmark Programs

	Size			Run time	
Program	Lines	Heap size (K)	Code size (K)	Non-gc (sec)	Gc (sec)
CW	5728	1107	894	22.74	3.09
Knuth-Bendix	491	2768	251	13.47	1.48
Lexgen	1224	2162	305	15.07	1.06
Life	111	1026	221	16.97	0.19
PIA	1454	1025	291	6.07	0.34
Simple	999	11571	314	25.58	4.23
VLIW	3207	1088	486	23.70	1.91
YACC	5751	1632	580	4.60	1.98

Table 2: Sizes of Benchmark Programs

Program	Inst Fetches	Reads (%)	Writes (%)	Partial Writes (%)	Assignments (%)	Nops (%)
CW	523,245,987	17.61	11.61	0.01	0.41	13.24
Knuth-Bendix	312,086,438	19.66	22.31	0.00	0.00	5.92
Lexgen	328,422,283	16.08	10.44	0.20	0.21	12.33
Life	413,536,662	12.18	9.26	0.00	0.00	15.45
PIA	122,215,151	25.27	16.50	0.00	0.00	8.39
Simple	604,611,016	23.86	14.06	0.00	0.05	7.58
VLIW	399,812,033	17.89	15.99	0.10	0.77	9.04
YACC	133,043,324	18.49	14.66	0.32	0.38	11.14

Table 3: Characteristics of benchmark programs

	Allocation	Escaping		Known		Callee Saved		Records		Other	
Program	(words)	%	Size	%	Size	%	Size	%	Size	%	Size
CW	56,467,440	4.0	4.12	3.3	15.39	67.2	6.20	19.5	3.01	6.0	4.00
Knuth-Bendix	67,733,930	37.6	6.60	0.1	15.22	49.5	4.90	12.7	3.00	0.1	15.05
Lexgen	33,046,349	3.4	6.20	5.4	12.96	72.7	6.40	15.1	3.00	3.7	6.97
Life	37,840,681	0.2	3.45	0.0	15.00	77.8	5.52	22.2	3.00	0.0	10.29
PIA	18,841,256	0.4	5.56	28.0	11.99	25.0	4.69	12.7	3.41	33.9	3.22
Simple	80,761,644	4.0	5.70	1.1	15.33	68.1	6.43	8.3	3.00	18.5	3.41
VLIW	59,497,132	9.9	5.22	6.0	26.62	61.8	7.67	20.3	3.01	2.1	2.60
YACC	17,015,250	2.3	4.83	15.3	15.35	54.8	7.44	23.7	3.04	4.0	10.22

Table 4: Allocation characteristics of benchmark programs

Table 4 gives the allocation statistics for each benchmark program. All allocation and sizes are reported in words. The *Allocation* column lists the total allocation done by the benchmark. The remaining columns break down the allocation by kind: closures for escaping functions, closures for known functions, closures for callee-save continuations[8], records, and others (includes spill records, arrays, strings, vectors, ref cells, store list records, and floating point numbers). For each allocation kind, the % column gives the total words allocated for objects of that kind as a percentage of total allocation and the *Size* column gives the average size in words, including the 1 word tag, of an object of that kind.

4.4 Metrics

Following the lead of recent work on memory subsystem performance, we state cache performance numbers in *cycles per useful instruction* (*CPI*). All instructions besides *nops* are considered useful. Unlike miss ratios, CPI numbers give an indication of how fast a program will run. On the down side, CPI numbers are machine dependent because actual penalties are used in their calculations.

Table 5 lists the penalties used in our simulations. These numbers are derived from the penalties for the DECStation 5000/200, but are similar to those in other machines of the same class. Writes have different penalties depending on whether or not subblock placement is being used, the block size (and thus the fetch size), and whether the writes hit or miss in the cache. For caches with subblock placement, write hits or misses have no penalty (besides write buffer related costs)[9]. For

[8] Closures for callee-save continuations can be trivially allocated on a stack in the absence of first class continuations.

[9] In an actual implementation, the penalty of a miss may be one cycle since unlike hits, the tag needs to be written

Task	Penalty (in cycles)
Non-page-mode write	5
Page-mode write	1
Page-mode flush	4
Read 16 bytes from memory	15
Read 32 bytes from memory	19
Refresh period	195
Refresh time	5
Write hit or miss (subblocks)	0
Write hit (16 bytes, no subblocks)	0
Write hit (32 bytes, no subblocks)	0
Write miss (16 bytes, no subblocks)	15
Write miss (32 bytes, no subblocks)	19

Table 5: Penalties of memory operations

caches without subblock placement, write hits have no penalty (besides write buffer related costs) but write misses cost 15 or 19 cycles (plus write buffer penalties) for block sizes of 16 and 32 bytes respectively. The read miss and instruction fetch miss penalty depends on the block size: it is 15 cycles for a block size of 16 bytes and 19 cycles for a block size of 32 bytes.

We used a DRAM page size of 4K in the simulation of page-mode writes. Page-mode flush is the number of cycles needed to flush the write pipeline after a series of page-mode writes.

TLB data is reported as (CPI – CPI of perfect memory subsystem[10]). This is the TLB contribution to the CPI. This metric is used instead of just CPI to allow us to present the measurements for all the benchmarks in one chart. A virtual memory page size of 4K was used in the simulations. The penalty of a TLB miss is 28 cycles[11].

5 Results and Analysis

In Section 5.1 we present a qualitative analysis of the memory behavior of programs compiled with SML/NJ. In Section 5.2 we list the cache and TLB configurations simulated and explain why they were selected. In Sections 5.3, 5.4, and 5.5 we present data for memory subsystem performance, write buffer performance, and TLB performance. In Section 5.6 we validate the simulations. In Section 5.7 we present an analytical model which allows us to extend the memory subsystem performance results to programs with different allocation behavior. In Section 5.8 we summarize the results.

to the cache after the miss is detected. This will not change our results since it adds at most 0.02–0.05 to the CPI of caches with subblock placement.

[10] The CPI of a perfect memory subsystem is the total number of instructions divided by the number of useful instructions.

[11] This is a weighted average of the various kinds of TLB misses under Mach 3.0 and is derived from the data in [46].

Write Policy	Write Miss Policy	Subblocks	Assoc	Block Size	Cache Sizes	Write Buffer
through	allocate	yes	1, 2	16, 32 bytes	8K–512K	1–6 deep
through	allocate	no	1, 2	16, 32 bytes	8K–512K	6 deep
through	no allocate	no	1, 2	16, 32 bytes	8K–512K	6 deep

Table 6: Cache organizations studied

5.1 Qualitative Analysis

Recall from Section 2 that SML/NJ uses a copying collector. The most important property of a copying collector with respect to memory subsystem behavior is that allocation initializes memory in an area that has not been touched since the last garbage collection. This means that for caches that are not large enough to contain the allocation area there will be a large number of write misses. The slowdown that the write misses translates into depend on the memory subsystem organization.

Recall from Section 4.3 that SML/NJ programs have the following important properties. First, they do few assignments; the majority of the writes are initializing writes. Second, programs do heap allocation at a furious rate: 0.1 to 0.22 words per instruction. Third, writes come in bunches because they correspond to initialization of a newly allocated area.

The burstiness of writes combined with the property of copying collectors mentioned above suggests that an aggressive write policy is necessary. In particular, writes should not stall the CPU. Memory subsystem organizations where the CPU has to wait for a write to be written through (or back) to memory *will perform poorly*. Even memory subsystems where the CPU does not need to wait for writes if they are issued far apart (e.g., 2 cycles apart in the HP 9000 series 700) may perform poorly due to the bunching of writes. This leads to two requirements on the memory subsystem. First, a write buffer or fast page mode writes are essential to avoid waiting for writes to memory. Second, on a write miss, the memory subsystem must avoid reading a cache block from memory if it is going to be written before being read. Of course, this requirement only holds for caches with a *write-allocate* policy. Subblock placement [30], a block size of 1 word, and the ALLOCATE instruction [38] can all achieve this. Since the effects on cache performance of these features are so similar, we talk just about subblock placement. For large caches, when the allocation area fits in the cache and thus there are few write misses, the benefit of subblock placement will be reduced.

5.2 Cache and TLB configurations simulated

The design space for memory subsystems is enormous. There are many variables involved and the dependencies between them are complex. Therefore we could study only a subset of the memory subsystem design space. In this study, we restrict ourselves to features found in *currently popular* RISC workstations. Exploration of more exotic memory subsystem features is left to future work (see Section 6). Table 6 summarizes the cache organizations simulated. Table 7 lists the memory subsystem organization of some popular machines.

We simulated only separate instruction and data caches (*i.e.*, no unified caches). While many current machines have separate caches (e.g., DECStations, HP 700 series), there are some exceptions (notably SPARCStations).

We simulated cache sizes of 8K to 512K. This range includes the primary caches of most current machines (see Table 7). We consider only one-way (direct mapped) and two-way set associative caches (with LRU replacement).

We simulated block sizes of 16 bytes and 32 bytes. Moreover, fetch size is kept the same as the block size: in particular, in caches with subblock placement, a read miss brings in the whole block, not just the subblock causing the miss. In effect, this is prefetching. Przybylski [39] notes that making the fetch size equal to the block size is a good choice with respect to memory subsystem

Architecture	Write Policy	Write Miss Policy	Write Buffer	Subblocks	Assoc	Block Size	Cache Size
DS3100 [19]	through	allocate	4 deep		1	4 bytes	64K
DS5000/200 [18]	through	allocate	6 deep	yes	1	16 bytes	64K
HP 9000 [43]	back	allocate	none	no	1	32 bytes	64K–2M
SPARCStation II [17]	through	no allocate	4 deep	no	1	32 bytes	64K

Note:

- SPARCStations have unified caches.
- Most HP 9000 series 700 caches are much smaller than 2M: 128K instruction cache and 256K data cache for models 720 and 730, and 256K instruction cache and 256K data cache for model 750.
- The DS5000/200 actually has a block size of four bytes with a fetch size of sixteen bytes. *This is stronger than subblock* placement since it has a full tag on every "subblock".
- The higher end HP 9000 machines (model 735 and above) provide a cache-control hint in their store instructions[11]. The hint can specify that a block will be overwritten before being read; this avoids the read if the write misses. The SML/NJ compiler may be able to extract much of the benefits of subblock placement from this feature.

Table 7: Memory subsystem organization of some popular machines

performance. Przybylski also notes that block sizes of 16 or 32 bytes optimize the read access time for the memory parameters used in the CPI calculations (see Table 5). Hereafter, whenever *subblock placement* is mentioned, it is assumed that the fetch size equals block size.

We report data only for write-through caches but the CPI for write-back caches *can be inferred* from the graphs for write-through caches. While write-through and write-back caches have identical misses, their contribution to the CPI may differ due to two reasons. First, a write hit or miss in a write-back cache may take one cycle more than in a write-through cache; unlike a write-through cache, a write-back cache must probe the tag *before* writing to the cache [27]. The graphs for write-through caches can be easily adjusted to account for this to obtain the graphs for write-back caches. For instance, if the program has w writes and n useful instructions, then the CPI for a write-back cache can be obtained by adding w/n to the CPI of the write-through cache with the same size and configuration. For VLIW w/n is 0.18. Second, write-through and write-back caches may have different write buffer penalties because they do writes to main memory with different frequencies and at different points. We expect the write buffer penalties for write-back caches to be smaller than those for write-through caches since writes to main memory are less frequent for write-back caches than for write-through caches. This difference between write-through and write-back caches is likely to be negligible since the write-buffer penalty is small even for write-through caches.

We varied write buffer depths from 1 to 6 entries for write-through caches with the *write allocate/subblock placement* organization. We also simulated memory subsystems with and without page-mode writes.

We simulated fully associative, unified TLBs from 1 to 64 entries with LRU replacement policy. Some machines (such as the HP 9000 series) have separate instruction and data TLBs. From Section 5.5 it is clear that for the benchmarks even small unified TLBs perform well.

Two of the most important cache parameters are *write allocate* versus *write no allocate* and *subblock placement* versus *no subblock placement*. Of these, the combination *write no allocate/subblock placement* placement offer no improvement over *write no allocate/no subblock placement* for cache performance. Thus, we did not collect data for the *write no allocate/subblock placement* configuration.

We restrict ourselves only to the first two levels of the memory hierarchy, which on most current machines corresponds to the primary cache and main memory. The results, however, are mostly applicable when the second level is a secondary cache and the cost of accessing the secondary cache is similar to the cost of accessing main memory in the *DECStation 5000/200*[12]. In such machines, there is a memory subsystem contribution to the CPI that we did not measure: a miss on the second level cache. Therefore the CPI obtained on these machines can be higher than that reported here.

We did not simulate the exotic features appearing on some newer machines, such as stream buffers, prefetching, scoreboarding, and victim caches. These features can reduce the number of cache misses and miss costs. Further work is needed to understand the impact of these features on the performance of heap allocation.

5.3 Memory Subsystem Performance

We present memory subsystem performance in summary graphs and breakdown graphs. Each summary graph summarizes the memory subsystem performance of one benchmark program for a range of cache sizes (8K to 512K), write-miss policies (write allocate or write no allocate), subblock placement (with or without), and associativity (1 or 2). Each curve in a summary graph corresponds to a different memory subsystem organization. There are two summary graphs for each program, one for a block size of 16 bytes and another for a block size of 32 bytes. Each breakdown graph breaks down the memory subsystem overhead into read misses, write misses (if there is a penalty for write misses), instruction fetch misses, write-buffer overhead, and partial-word write overhead for one configuration in a summary graph. The write-buffer depth in these graphs is fixed at 6 entries.

In this section we present only the summary graphs for VLIW (Figure 2). The summary graphs for other programs are similar and are given in Appendix A. Figures 3, 4, and 5 are the breakdown graphs for VLIW for the 16 byte block size configurations; the remaining breakdown graphs for *VLIW* are *similar and omitted* for conciseness. The breakdown graphs for the other benchmarks are similar (and predictable from the summary graphs) and are thus omitted for the same reason[13].

In the summary graphs, the *nops* curve is the base CPI: the total number of instructions executed divided by the number of useful (not nop) instructions executed; this corresponds to the CPI for a perfect memory subsystem[14]. For the breakdown graphs, the *nop* area is the CPI contribution of nops; *read miss* is the CPI contribution of read misses; *write miss* is the CPI contribution of write misses (if any), *inst fetch miss* is the CPI contribution of instruction fetch misses; *write buffer* is the CPI contribution of the write buffer; *partial word* is the CPI contribution of partial-word writes.

The 64K point on the *write alloc, subblock, assoc=1* curves corresponds closely to the DECStation 5000/200 memory subsystem.

In the following subsections we describe the impact of write-miss policy and subblock placement, associativity, block size, cache size, write buffer, and partial-word writes on the memory subsystem performance of the benchmark programs.

5.3.1 Write Miss Policy and Subblock Placement

From the summary graphs, it is clear that the best cache organization we studied is *write allocate/subblock placement*; it substantially outperforms all other configurations. Surprisingly, for sufficiently large caches with the *write allocate/subblock placement* organization, the memory subsystem performance of SML/NJ programs is acceptable: the overhead due to data cache misses ranges from 3% to 13% (arithmetic mean 9%) for 64K direct mapped caches[15] and 1% to 13% (arithmetic mean 9%) for 32K two-way associative caches. The memory subsystem performance of

[12] For instance, Borg et al. [10] use 12 cycles as the latency for going to the second level cache and 200-250 cycles for going to memory.

[13] The full set of graphs is available via anonymous ftp from ibis.cs.umass.edu in pub/memory-subsystem.

[14] *nops* constitute between 5.9% and 15.4% of all instructions executed for the benchmarks (see Section 4.3).

[15] Recall that this corresponds to the DECStation 5000/200 memory subsystem.

SML/NJ programs on the DECStation 5000/200 is comparable to that of C and Fortran programs [12]; Chen and Bershad find that the data cache overhead of C and Fortran programs ranges from less than 1% to 66%, with an arithmetic mean of 8%[16]. It is worth emphasizing that the memory subsystem performance of SML/NJ programs is *good* on some current machines *despite the very high miss rates*; for a 64K *write allocate/no subblock placement* organization with a block size of 16 bytes, the write miss and read miss ratios for `VLIW` are 0.23 and 0.02 respectively.

Recall that in Section 5.1 we argued that the benefit of subblock placement would be substantial, but that the benefit would decrease for larger caches. The summary graphs indicate that the reduction in benefit is not substantial even for 128K cache sizes; however, the benefit of subblock placement decreases sharply for larger caches for six of the benchmark programs. This suggests that the allocation area size of six of the benchmark programs is 256K to 512K.

The performance of *write allocate/no subblock* is almost identical to that of *write no allocate/no subblock* (**Leroy** is an exception)[17]. This suggests that an address is being read soon after being written; even in an 8K cache, an address is read after being written before it is evicted from the cache (if it was evicted from the cache before being read, then *write allocate/no subblock* would have inferior performance). The only difference between these two schemes is *when* a cache block is read from memory. In one case, it is brought in on a write miss; in the other, it is brought in on a read miss. Because SML/NJ programs allocate sequentially and do few assignments, a newly allocated object remains in the cache until the program has allocated another `C` bytes, where `C` is the size of the cache. Since the programs allocate 0.4-0.9 bytes per instruction, our results suggest that a read of a block occurs within 9K-20K instructions of its being written.

5.3.2 Changing Associativity

From Figure 2 we see that increasing associativity improves all organizations. However the improvement in going from one-way to two-way set associativity is much smaller than the improvement obtained from subblock placement: in most cases, it improves the CPI by less than 0.1. The maximum benefit from higher associativity is obtained for small cache sizes (less than 16K). However, increasing associativity may increase CPU cycle time and thus the improvements may not be realized in practice [25].

From Figures 3, 4, and 5 we see that higher associativity improves the instruction cache performance but has little or no impact on data cache performance. Surprisingly, for direct mapped caches (Figures 3 (a), 4 (a), and 5 (a)) the instruction cache penalty is substantial for 128K or smaller caches. For caches with subblock placement, the instruction cache penalty can dominate the penalty for the memory subsystem. The improvement observed in going to a two-way associative cache suggests that a lot of the penalty from the instruction cache is due to conflict misses and that from the data cache is due to capacity misses: the data cache is simply not big enough to hold the working set. When the benchmark programs are examined, the performance of the instruction cache is not surprising: the code consists of small functions with frequent calls, which lowers the spatial locality. Thus, the chances of conflicts are greater than if the instructions had strong spatial locality.

5.3.3 Changing Block Size

From Figure 2 we see that increasing block size from 16 to 32 bytes also improves performance. For the *write allocate* organizations, an increased block size decreases the number of write misses caused by allocation. When the allocation area does not fit in the cache, doubling the block size can halve the write-miss rate. Thus, larger block sizes improve performance when there is a penalty

[16] Chen and Bershad use Cycles/Instruction rather than Cycles/Useful Instruction which lowers their memory subsystem overhead.

[17] The difference between *write allocate/no subblock* and *write no allocate/no subblock* is so small in most graphs that the two curves overlap.

for a write miss [30]. In particular, larger block sizes have little to offer to caches with *write allocate/subblock placement*. From Figure 2 we see that the *write no allocate* organizations benefit just as much from larger block size as *write allocate/no subblock placement*; this suggests that the spatial locality of the reads is comparable to that of the writes.

Note that subblock placement improves performance more than even two-way associativity and 32 byte blocks combined.

5.3.4 Changing Cache Size

Three distinct regions of performance can be identified for cache sizes. The first region corresponds to the range of cache sizes when the allocation area does not fit in the cache (i.e., allocation happens in an area of memory which is not cache resident). For most of the benchmarks, this region corresponds to cache sizes of less than 256K (for **Simple** and **Knuth-Bendix** this region extends beyond 512K). In this region, increasing the cache size uniformly improves performance for all configurations. However, the performance improvement from doubling the cache size is small.

From the breakdown graphs we see that in the first region the cache size has little effect on the data cache miss contribution to CPI. Most of the improvement in CPI that comes from increasing the cache size is due to improved performance of the instruction cache. As with associativity, cache sizes have interactions with the cycle time of the CPU: larger caches can take longer to access. Thus, improvement due to increasing the cache size may not be achieved in practice.

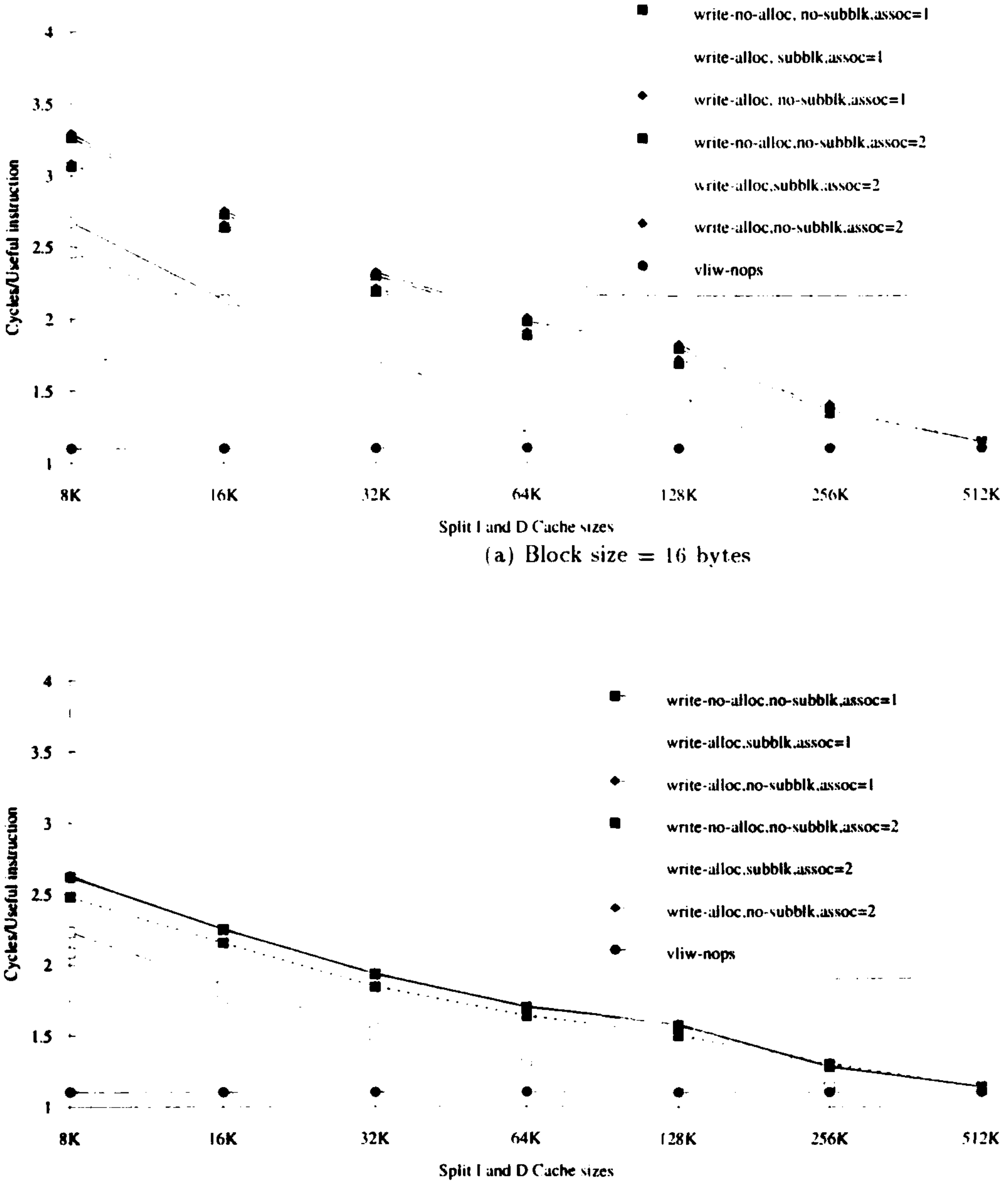
The second region ranges from when the allocation area begins to fit in the cache until the allocation area fits in the cache. For most of the benchmarks (once again excepting **Simple** and **Knuth-Bendix**), this region corresponds to cache sizes in the range 256K to 512K[18]. In this region, increasing the cache size sharpiy improves the data cache performance for memory organizations without subblock placement. However, increasing the cache size in this region has little to offer for instruction cache performance because the instruction cache miss penalty is already low at this point.

The third region corresponds to cache sizes when the allocation area fits in the cache. For five of the benchmarks, this region corresponds to caches larger than 512K (for **Lexgen**, **Knuth-Bendix**, and **Simple** this region starts at larger cache sizes). In this range, increasing the cache size has little or no impact on memory subsystem performance because everything remains cache resident and thus there are no capacity misses to eliminate.

5.3.5 Write Buffer and Partial-Word Write Overheads

From the breakdown graphs we see that the write buffer and partial word write contribution to the CPI is negligible. A six deep write buffer coupled with page-mode writes is sufficient to absorb the bursty writes. As expected, memory subsystem features which reduce the number of misses (such as higher associativity and larger cache sizes) also reduce the write buffer overhead.

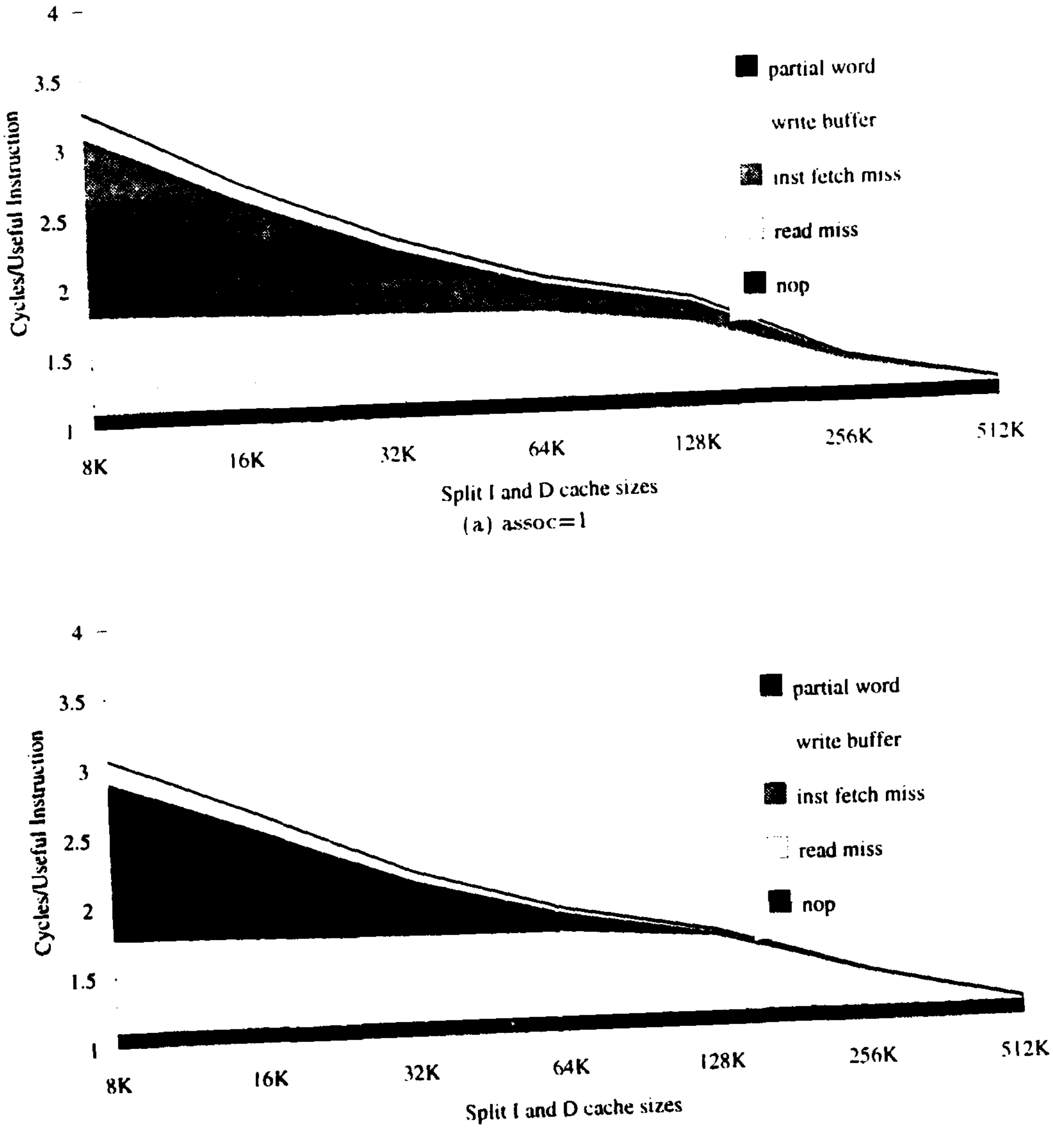
[18] For **Lexgen** this region extends a little beyond 512K.



(a) Block size = 16 bytes

(b) Block size = 32 bytes

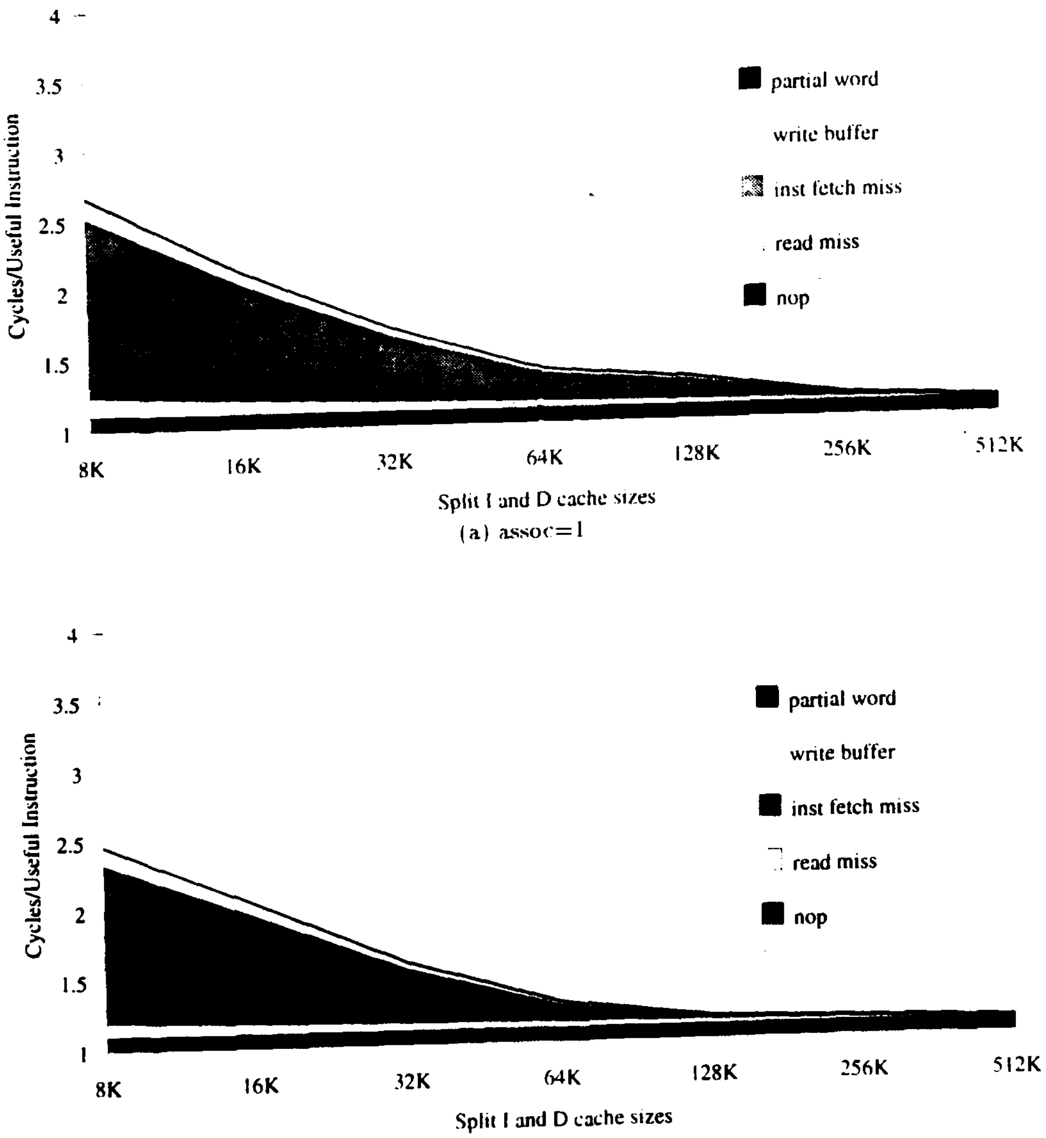
Figure 2: VLIW summary



(a) assoc=1

(b) assoc=2

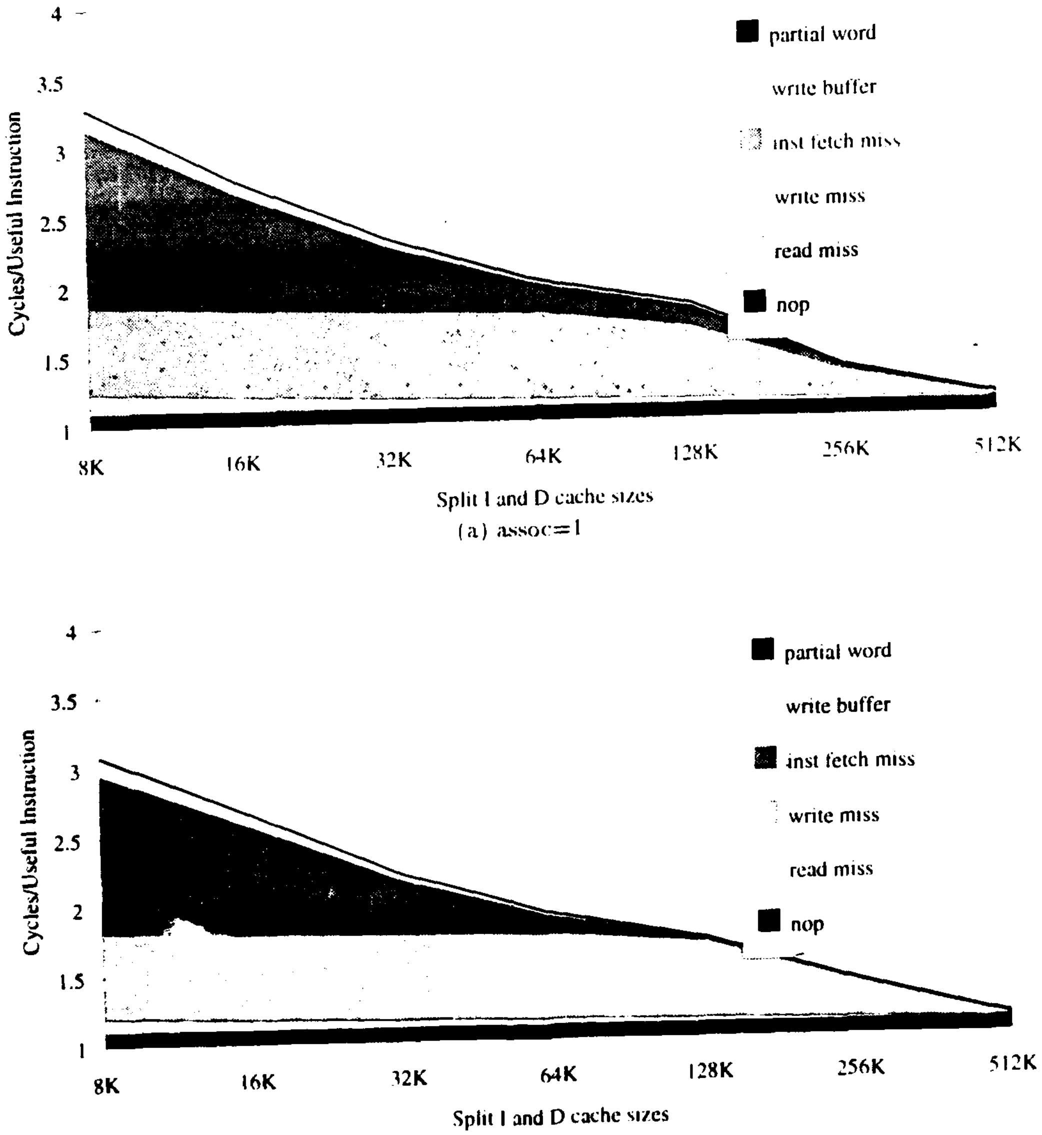
Figure 3: VLIW breakdown, write no alloc, no subblk, block size=16



(a) assoc=1

(b) assoc=2

Figure 4: VLIW breakdown, write alloc, subblk, block size=16



(a) assoc=1

(b) assoc=2

Figure 5: VLIW breakdown, write alloc, no subblk, block size=16

5.4 Write-buffer depth

In Section 5.3.5 we showed that a six-deep write buffer coupled with page-mode writes was able to absorb the bursty writes in SML/NJ programs. In this section we explore the impact of write buffer depth on the write-buffer contribution to CPI. Since the speed at which the write buffer can retire writes depends on whether or not the memory subsystem has page-mode writes, we conducted two sets of experiments. In the first set, we simulated a memory subsystem with page-mode writes and varied the write-buffer depth from 1 to 6. In the second set, we simulated a memory subsystem without page-mode writes and varied the write-buffer depth from 1 to 6. We conducted this study for two of the larger benchmarks: CW and VLIW. We fixed the block size at 16 bytes and the write miss policy at *write allocate/subblock placement*.

Figure 6 gives the write buffer overheads for VLIW with caches of associativity one and two and in a memory subsystem with page-mode writes; Figure 7 does the same in a memory subsystem without page-mode writes. The graphs plot the CPI contribution of the write buffer against cache size; there is one curve for each write-buffer depth. Graphs for CW are omitted for space considerations. Increasing the cache size or associativity reduces the number of read and instruction fetch misses, and thus reduces the number of main memory transactions. This reduces the write-buffer contribution to the CPI in four ways:

1. The write buffer has more cycles to retire its entries and hence the *write buffer full* stalls occur less frequently[19].

2. In the memory subsystem with page-mode writes, the main memory is thrown out of page mode less frequently, allowing the write buffer to retire writes quickly[20]. This reduces the *write buffer full* stalls.

3. Since there are fewer reads to main memory, the number of times a read to main memory needs to wait for a write to finish is less, thus reducing the *main memory busy* delays.

4. Since there are fewer reads to main memory, a read to main memory conflicts with a write buffer entry less frequently, thus reducing the *write buffer conflict* delays.

In memory subsystems with page-mode writes (Figure 6), the difference between the CPI contribution of a one-deep write buffer and a six-deep write buffer is less than 0.05. This is surprisingly small considering the burstiness of the writes. This is due to the effectiveness of page-mode writes: an example illustrates this:

Suppose that a SML/NJ program is allocating (and initializing) an object which is 4 words in size and that the write buffer is one deep. Further suppose that the write buffer is empty and that the instructions doing the allocation all hit in the instruction cache. The first write does not stall the CPU since the write buffer is empty. The next write comes one cycle later, finds a full write buffer, and thus stalls the CPU. After 4 cycles (see penalties in Table 5), the write is queued up in the write buffer. This write, however is highly likely to be on the same DRAM page as the previous write (since it is to the next address) and will therefore take only one cycle to complete. All subsequent writes to initialize this object find an empty write buffer since they all complete in one cycle due to page-mode writes.

As noted above, all the writes to initialize an object are likely to be on the same page and can thus take advantage of page-mode writes. Due to sequential allocation, it is likely that writes to initialize objects allocated one after another will also be on the same DRAM page. Thus, in the best case (with no read misses and refreshes), a *write buffer full* delay will happen only once per N words of allocation, where N is the size of the DRAM page. Thus, the write buffer depth has little performance impact on SML/NJ programs if the memory subsystem has page-mode writes.

[19] Recall that a write buffer uses free memory cycles to retire its writes.

[20] Recall that reads throw main memory out of page mode.

To confirm this explanation, we measured the probability of two consecutive writes being on the same DRAM page. This probability (averaged over the benchmarks) was 96%.

The small impact of write buffer depth on performance does not imply that a write buffer is useless if the memory system has page-mode writes. Instead, it says that a write buffer offers little performance improvement in a memory subsystem with page-mode writes if the programs have ***strong spatial locality*** in the writes, and the majority of the reads and instruction fetches hit in the cache. ***Strong spatial locality*** means that the probability that two consecutive writes are to the same DRAM page is very high.

Write-buffer depth is however important if the memory subsystem does not have page-mode writes (Figure 7). A six-deep write buffer performs substantially better than a one-deep write buffer in a memory system without page-mode writes.

5.5 TLB Performance

Figure 8 gives the TLB miss contribution to the CPI for each benchmark program. We see that CPI contribution of TLB misses falls below 0.01 for all our programs for a *64 entry unified* TLB; for half the benchmarks, it is below 0.01 even for a 32 entry TLB.

5.6 Validation

To validate our simulations, we ran each of the benchmarks five times on a DECStation 5000/200 (running Mach 2.6) and measured the *user time* for each run. The programs were run on a lightly loaded machine but not in single-user mode. The simulations with *write allocate/subblock placement*, 64K direct-mapped caches, 16 byte blocks, and 64 entry TLB corresponds closely to the DECStation 5000/200 with the following important differences:

- The simulations ignored the effects of context switches and system calls. Thus, actual program runs suffered more data and instruction cache misses than those reported by the simulations [36].

- The simulations assumed a *virtual address=physical address* mapping. Kessler and Hill [29] show that random mapping (as used in the actual runs) can have many more conflict misses than a careful mapping (such as that assumed by the simulations). Thus, the actual runs probably suffered more conflict misses than those reported by the simulations.

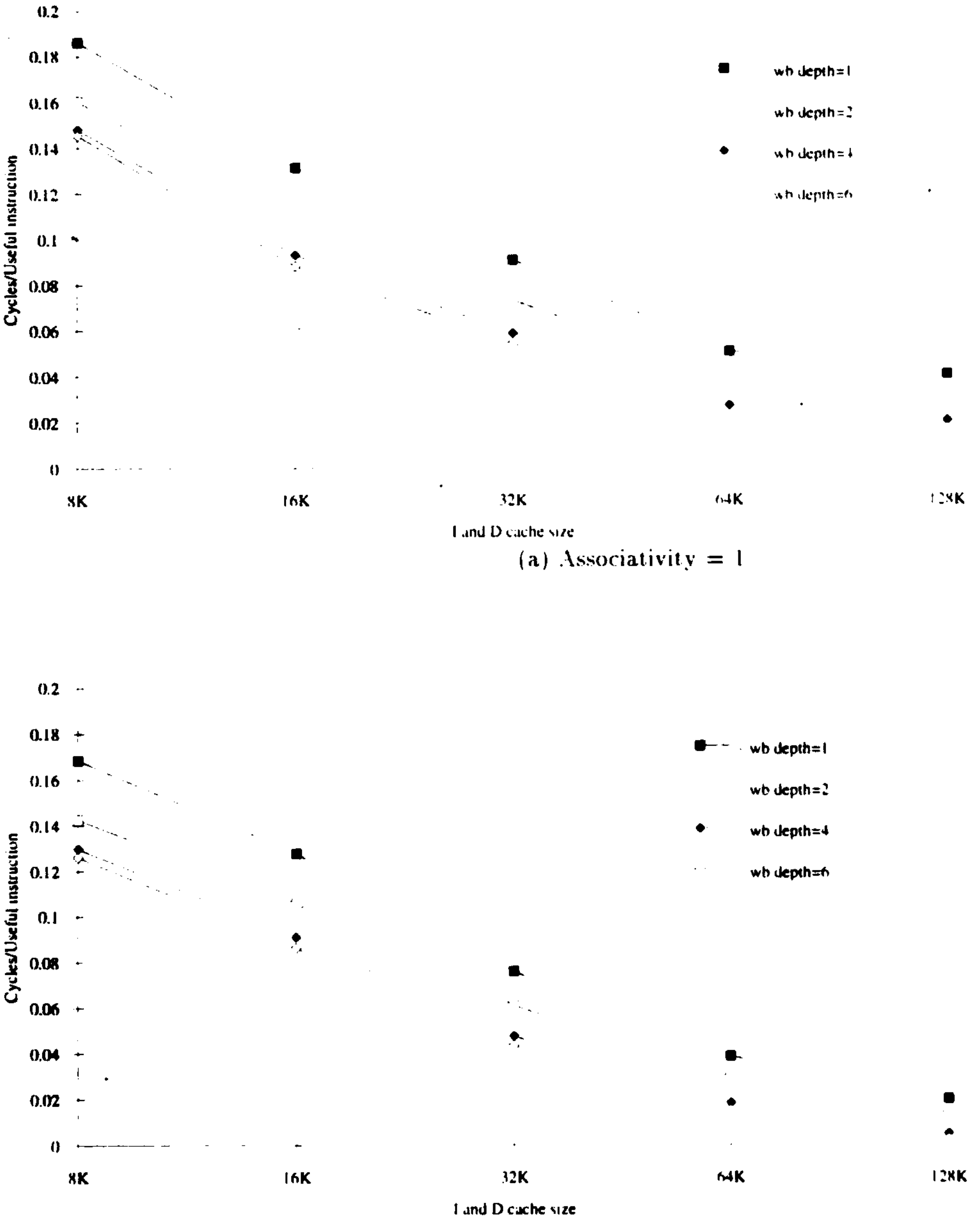
- The simulations assumed that all instructions take exactly one cycle (plus memory subsystem overhead). Some of the benchmarks do multiplications and divisions (both of which take more than one cycle). Thus, the actual program runs may take more cycles to complete than the cycles predicted by the simulations.

In order to minimize the memory subsystem effects of the virtual to physical mapping and context switches, we took the minimum CPI of the five runs for each program and compared it to the CPI obtained via simulations. We present our findings in Table 8: *Measured (sec)* is the user time of the program in seconds; *Measured CPI* is the CPI obtained from the measured time; *Simulated CPI* is the CPI obtained from the simulations; *Difference* is the difference between the measured CPI and the simulated CPI; *Discrepancy* is the difference as a percentage of measured CPI.

Table 8 shows that with the exception of `PIA` and `VLIW`, the discrepancy is small (*i.e.*, less than 10%); the actual runs validate the simulations. The discrepancy in `PIA` and `VLIW` is due to the significant number of multi-cycle instructions they execute[21]. Table 9 lists the multi-cycle instructions executed by each program[22]. *Total* is the percentage of instructions which are divisions,

[21] In this section, multi-cycle instructions refer to integer multiplication and division, and floating point operations.

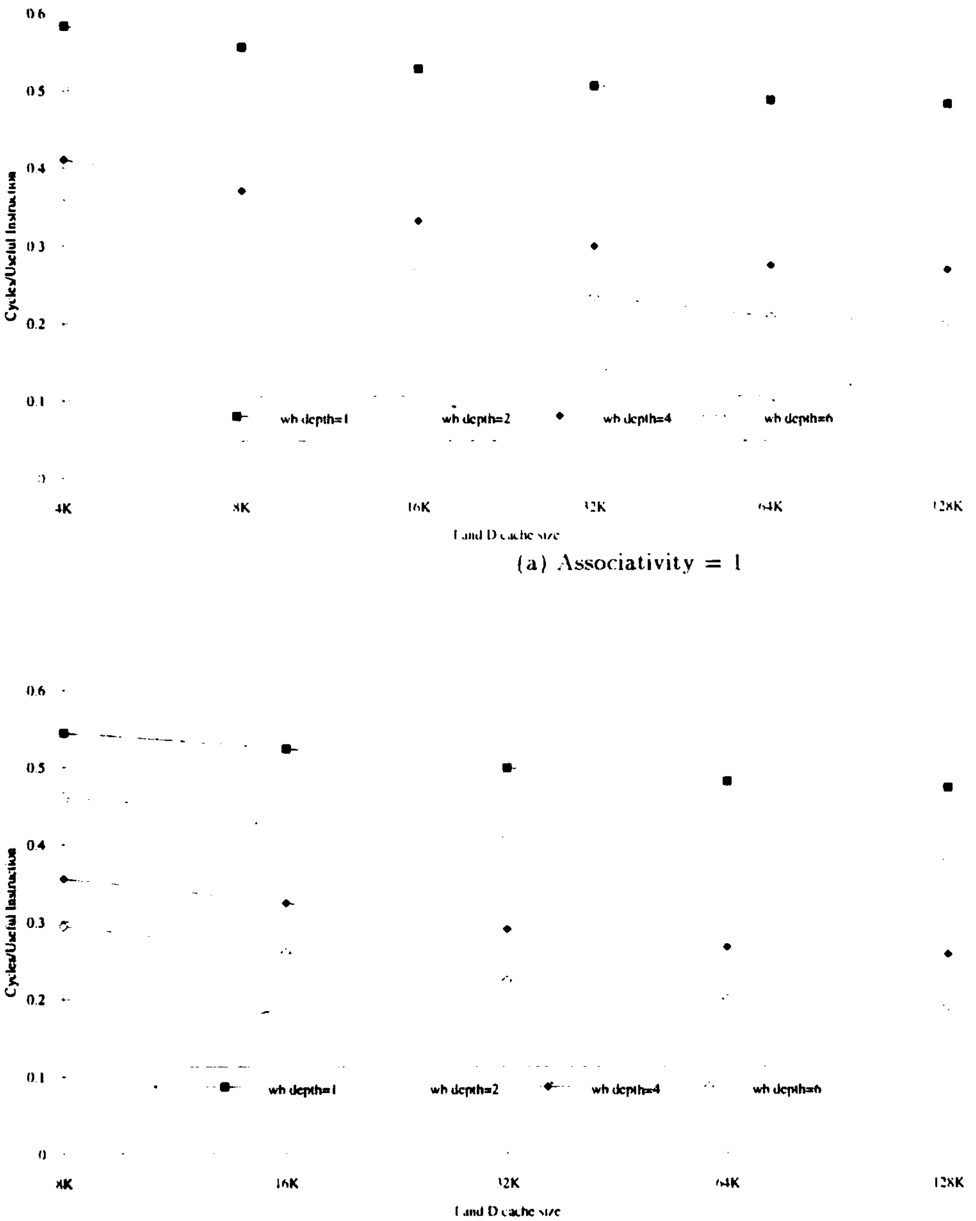
[22] SML/NJ uses only the "double" versions of each floating point instruction.



(a) Associativity = 1

(b) Associativity = 2

Figure 6: Write buffer CPI contribution for VLIW. With page-mode writes



(a) Associativity = 1

(b) Associativity = 2

Figure 7: Write buffer CPI contribution for VLIW, Without page-mode writes

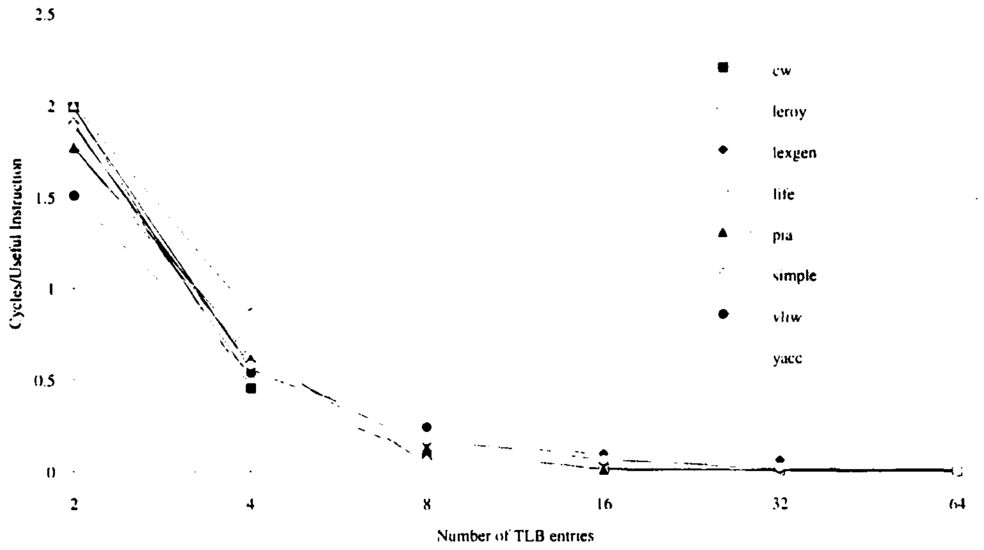


Figure 8: TLB contribution to CPI

Program	Measured (sec)	Measured CPI	Simulated CPI	Difference	Discrepancy (%)
CW	25.83	1.42	1.39	0.03	2.18
Knuth-Bendix	14.95	1.27	1.21	0.06	5.22
Lexgen	16.13	1.40	1.31	0.09	6.29
Life	17.16	1.23	1.21	0.02	1.19
PIA	6.41	1.43	1.18	0.25	17.62
Simple	29.81	1.33	1.21	0.12	9.03
VLIW	25.61	1.76	1.39	0.37	20.77
YACC	6.58	1.39	1.36	0.03	2.20

Table 8: Measured versus Simulated

multiplications, floating point additions, or floating point subtractions; *I Div* and *I Mul* are the percentages of integer division and multiplication respectively; *F Add*, *F Sub*, *F Div*, *F Mul* are the percentages of floating point additions, subtractions, divisions, and multiplications respectively.

The actual impact of multi-cycle instructions on CPI can be determined only by simulations. This is because on a DECStation 5000/200, the CPU does not need to wait after issuing a multi-cycle instruction. However, if the CPU tries to read the result of a multi-cycle instruction, it stalls until that instruction is complete. Moreover, the number of cycles needed for a floating point instructions depends on what other operations are currently in progress in the floating point coprocessor. Table 10 gives the latencies (in cycles) for the different multi-cycle instructions. The cycles for the floating point multiplication and division are lower bounds.

To test whether multi-cycle instructions could explain the high discrepancies in PIA and VLIW, we added the overhead of multi-cycle instructions to the simulated CPI assuming that all multi-cycle instructions stalled the CPU for the cycles listed in Table 10. This yielded a simulated CPI of 1.41 for PIA and 1.59 for VLIW. This reduced the discrepancy to 1.4% for PIA and 9.7% for VLIW.

On examining the assembly code generated for PIA, we found that the distance between multi-cycle instructions and use of their results varied significantly. Moreover, in many instances the assembly code had bunches of multiplications and divisions; these cause resource conflicts in the floating-point coprocessor thus causing them to have longer latencies than those in Table 10. Therefore, without simulating multi-cycle instructions, we cannot determine their exact penalty in PIA.

Program	Total	I Div	I Mul	F Add	F Sub	F Div	F Mul
CW	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Knuth-Bendix	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Lexgen	0.04	0.02	0.02	0.00	0.00	0.00	0.00
Life	0.00	0.00	0.00	0.00	0.00	0.00	0.00
PIA	4.08	0.00	0.00	1.30	0.38	0.84	1.56
Simple	1.67	0.00	0.50	0.30	0.14	0.06	0.67
VLIW	0.95	0.32	0.63	0.00	0.00	0.00	0.00
YACC	0.01	0.01	0.00	0.00	0.00	0.00	0.00

Table 9: Multi-cycle instructions as a percentage of instruction count

	Integer	Floating Point
Multiplication	13	4
Division	36	18
Addition	–	1
Subtraction	–	1

Table 10: Multi-cycle instruction cost on a DECStation 5000/200

However, a simple calculation shows that even if each multi-cycle instruction stalls the CPU for half the time reported in Table 10, the discrepancy falls well below 10%. Thus, multi-cycle instructions can explain the discrepancy for PIA.

From profiling VLIW we found that the vast majority of the multi-cycle instructions came from one routine, mod, in the SML/NJ standard library. On examining the assembly code for mod, we found that the results of the multiplications were used immediately, and the results of the divisions were used either immediately or one instruction later. Thus each multiplication stalled the CPU for 13 cycles and each division stalled the CPU for 35[23] or 36 cycles. Thus, it is reasonable to use the numbers in Table 10 to compute CPI overhead of multi-cycle instructions. Thus, multi-cycle instructions can explain the discrepancy for VLIW.

5.7 Extending the results

Section 5.3 demonstrated that heap allocation can have a significant memory subsystem cost if it is not possible to allocate a new object directly into the cache. In this section, we present and evaluate an analytic model which predicts the memory subsystem cost due to heap allocation when this is the case. This model formalizes the intuition presented in Section 5.1. It allows us to predict the memory subsystem cost due to heap allocation when block sizes, miss penalties, or program heap allocation rates change. We use the model to speculate about the memory subsystem cost of heap allocation for caches without subblock placement if SML/NJ were to use a simple stack.

[23] Assuming the instruction (always arithmetic) between the division and use of its result hits in the cache.

5.7.1 An analytic model

Recall that heap allocation with copying garbage collection typically allocates memory which has not been touched in a long time, and thus is unlikely to be in the cache. This is especially true when the allocation area does not fit in the cache. Thus, when newly allocated memory is initialized, write misses occur. The rate of write misses depends upon the allocation rate and the block size. Given the rate of write misses, we can calculate the memory subsystem cost, C, due to heap allocation.

$$\begin{aligned} a &= \text{allocation rate (words/useful instruction)} \\ b &= \text{block size (words)} \\ r_p &= \text{read miss penalty (cycles)} \\ w_p &= \text{write miss penalty (cycles)} \end{aligned}$$

Then under the assumption that the allocation area does not fit in the cache, *i.e.* initializing writes miss,

$$C_{\text{write alloc}} = w_p * a/b$$

The cost of allocating one word on the heap, A, will be

$$A_{\text{write alloc}} = w_p \; / \; b$$

Note that depending on the cache organization, the write miss penalty may be 0.
Under the additional assumption that programs touch allocated data soon after it is allocated,

$$\begin{aligned} C_{\text{write no alloc}} &= r_p * a/b \\ A_{\text{write no alloc}} &= r_p/b \end{aligned}$$

The cost of heap allocation should account for the difference in simulated CPIs when the write miss policy is varied for the SML/NJ benchmarks, since the benchmarks do so few assignments. That is,

$$\begin{aligned} C_{\text{write alloc/no subblock}} &\approx CPI_{\text{write alloc/no subblock}} - CPI_{\text{write alloc/subblock}} \\ C_{\text{write no alloc/no subblock}} &\approx CPI_{\text{write no alloc/no subblock}} - CPI_{\text{write alloc/subblock}} \end{aligned}$$

Table 11 shows the average percentage difference between the cost of heap allocation, C, and the differences in the CPIs. The percentage difference for write allocate/no subblock, D, was calculated as

$$\begin{aligned} CPI_{\text{diff}} &= CPI_{\text{write alloc/no subblock}} - CPI_{\text{write alloc/subblock}} \\ D_{\text{write alloc/no subblock}} &= \left| \frac{C_{\text{write alloc/no subblock}} - CPI_{\text{diff}}}{CPI_{\text{diff}}} \right| \end{aligned}$$

The percentage difference for *write no alloc/no subblock* was calculated similarly. We fixed the block size to be 16 bytes. Recall that the miss penalties are $w_p = r_p = 15$. We calculated the allocation rates (Table 12) for programs by using the allocation information from Table 4 and instruction counts from Table 3. The average was the arithmetic mean. The average difference when the allocation area does not fit in the cache (128K or less) is small (2-32%). When the assumption that the allocation area does not fit in the cache is violated, the model is inaccurate, as expected. The percentage difference heads towards infinity as CPI_{diff} becomes very small. Thus, this model can be used to predict the memory subsystem cost of heap allocation only for small cache sizes.

Cache size (Kilobytes)	$D_{\text{write no alloc/no subblock}}$ (%)	$D_{\text{write alloc/no subblock}}$ (%)
8K	7.12	2.4
16K	6.84	2.2
32K	7.02	2.2
64K	10.8	5.7
128K	31.8	23.5
256K	128.8	111.4
512K	1847.7	1746.2

Table 11: Percent difference between analytical model and simulations

Program	Allocation rate including callee-save conts. (words/useful instruction)	Allocation rate excluding callee-save conts. (words/useful instruction)
CW	0.12	0.04
Knuth-Bendix	0.23	0.12
Lexgen	0.11	0.03
Life	0.11	0.02
PIA	0.17	0.13
Simple	0.14	0.05
VLIW	0.16	0.06
YACC	0.14	0.07
Median	0.14	0.05

Table 12: Allocation rate for benchmarks, including and excluding callee-save continuations, which can be stack-allocated.

Program	C
	(cycles/instruction)
CW	0.15
Knuth-Bendix	0.44
Lexgen	0.12
Life	0.09
PIA	0.47
Simple	0.17
VLIW	0.23
YACC	0.24

Table 13: Assuming procedure activation records are stack allocated in SML/NJ, this table presents the expected memory subsystem cost of heap allocation for caches without subblock placement

5.7.2 SML/NJ with a stack

We can use this model to speculate about the memory subsystem cost of heap allocation in SML/NJ when a stack is used. In the absence of first-class continuations, which the benchmarks do not use, callee-save continuations can be easily stack-allocated. The callee-save continuations correspond to procedure activation records. Table 12 shows that stack-allocating callee-save continuations would greatly reduce the allocation rate of the benchmarks.

Assuming only continuations are stack-allocated, Table 13 presents an estimate of the memory subsystem cost of heap allocation for caches that do not have subblock placement and are too small to hold the allocation area. The block size is 16 bytes, the read miss penalty 15 cycles, and the write miss penalty for the no-subblock caches 15 cycles.

This is an upper bound estimate of expected memory subsystem cost of heap allocation with a stack because it may be possible to stack-allocate additional objects [31]. We see that even with a simple stack, the memory subsystem costs due to heap allocation for caches without subblock placement will probably be significant for SML/NJ programs.

5.8 Summary of Results

Contrary to what other researchers have speculated, we have found that the memory subsystem performance of SML/NJ is quite good on some real machines. Of the cache organization parameters we studied, *write allocate/subblock placement* with a subblock size of 1 word is most important for good performance of SML/NJ programs. However, small caches perform badly for all cache organizations. Also, DECStations are the only machines whose caches have subblock placement with a subblock size of 1 word; thus, the memory subsystem performance of SML/NJ programs is bad on most current machines.

Higher associativity and larger block sizes also improve performance but the improvement is not as significant as that offered by subblock placement. Larger cache sizes also improve performance, but for cache sizes up to 128K the improvement is small. For six of the benchmarks, increasing the cache sizes beyond 128K allows the allocation area to fit in the cache; thus increasing the cache size beyond 128K can be profitable.

Most surprisingly, higher associativity and larger cache sizes (up to 128K) have little effect on the performance of the data cache; most of the overall improvement observed is in the instruction cache. The bad locality of the instructions due to small functions and frequent calls leads to many conflict misses in the instruction cache, which can be alleviated by going to a larger cache size or higher associativity.

We found fast page mode writes to be very effective in absorbing the bursty writes of SML/NJ programs. In memory subsystems with page-mode writes, the write-buffer depth was not important: a one-deep write buffer performed almost as well as a six-deep write buffer. In memory subsystems without page-mode writes, the write buffer-depth was important: a one-deep write buffer performed much worse than a six-deep write buffer.

Finally, we found the penalty due to TLB misses to be small for TLBs with 32 or more entries.

6 Future Work

We suggest three directions in which this study can be extended:

- measuring the impact of other architectural features not explored in this work,
- measuring the impact of different compilation techniques, and
- measuring other aspects of programs.

Regarding *architectural features*, there is a need to explore memory subsystem performance of heap allocation on newer machines. As CPUs get faster relative to main memory, memory subsystem performance becomes even more crucial to good performance. To address the increasing discrepancy between CPU speeds and main memory speeds, newer machines, such as Alpha workstations [20], often have features such as secondary caches, stream buffers, and register scoreboarding.

Secondary caches improve performance by *reducing accesses to main memory*. Stream buffers and scoreboarding improve performance by reducing the latency of cache misses. The impact of these features on memory subsystem performance can be determined only by simulations. Previous work has addressed at least two of the features in isolation: Short and Levy [42], Borg *et al.* [10], and Przybylski [39] study two-level caches. Jouppi [26] studies stream buffers, and Chen and Baer [13] study scoreboarding. However, we are not aware of any published work which has studied a memory subsystem with all (or a combination) of these features. Also, we are not aware of any work evaluating the impact of these features on heap allocation.

Regarding *different compilation techniques*, the impact of stack allocation is worth measuring. A stack reduces heap allocation (which performs badly on most memory subsystem organizations) in favor of stack allocation (which can have good cache locality since it focuses most of the references to a small part of memory, namely the top of the stack). For SML/NJ programs, the majority of heap allocated objects can be allocated on the stack (Table 4). Therefore stack allocation can substantially improve performance of SML/NJ programs on memory organizations without subblock placement or with small cache sizes. However, stack allocation can slow down exceptions, first-class continuations, and threads. A careful study is needed to evaluate the pros and cons of doing stack allocation. We are currently working on this.

Regarding measuring *other aspects of programs*, several areas seem promising for future work:

1. Measuring the impact of different garbage collection algorithms on cache performance. Some work has already been done on this but more needs to be done (see Section 3).

2. Measuring the impact of changing various garbage collector parameters (such as allocation area size) on cache performance. We are currently working on this.

3. Measuring the cost of various operations related to garbage collection: tagging, store checks, and garbage collection checks. A preliminary study of this is reported in [45].

4.. Measuring the impact of optimizations on cache performance. Of special interest here is the effect of function inlining. We are currently working on this.

7 Conclusions

We have studied the memory subsystem performance of heap allocation with copying garbage collection, a general automatic storage management technique for modern programming languages. Heap allocation is useful for implementing language features such as list-processing, higher-order functions, and first-class continuations where objects may have indefinite extent. However, heap allocation is widely believed to have poor memory subsystem performance [38, 48, 49, 50]. This belief is based on the high (write) miss ratios that occur when new objects are allocated and initialized.

We studied the memory subsystem performance of mostly-functional SML programs compiled with the SML/NJ compiler. These programs heap allocate at intensive rates. They use heap-only allocation: all allocation, including activation records, is done on the heap. We simulated a wide variety of memory subsystems typical of current workstations.

To our surprise, we found that heap allocation performed well on some memory subsystems. In particular, on an actual machine (the DECStation 5000/200), the memory subsystem performance of heap allocation was good. However, heap allocation performed poorly on most memory subsystem organizations. The memory subsystem property crucial for achieving good performance was the ability to allocate and initialize a new object into the cache without a penalty. This can be achieved by having subblock placement or a cache large enough to hold the allocation area, along with fast page-mode writes or a sufficiently deep write buffer.

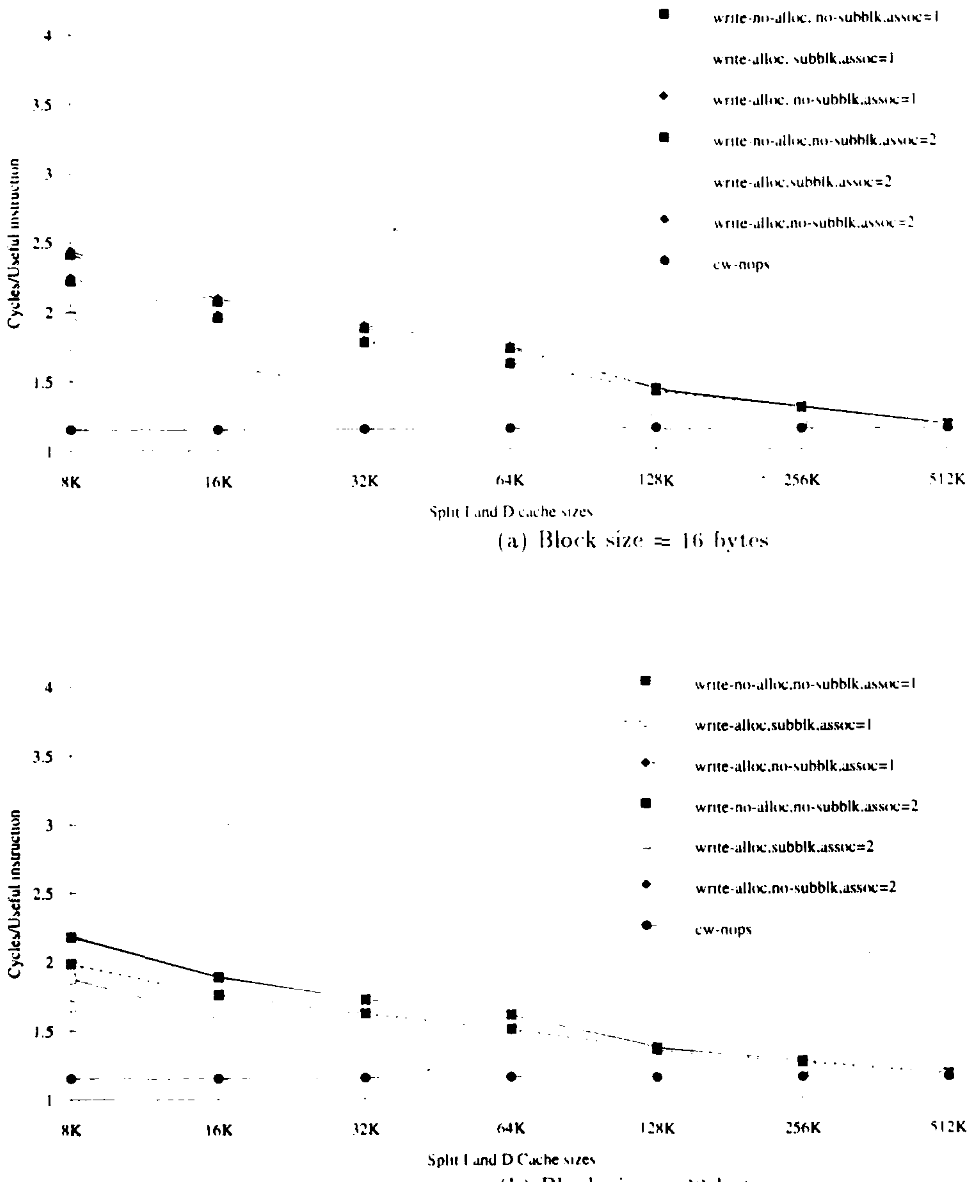
We found for caches with subblock placement, the arithmetic mean of the data cache penalty was under 9% for 64K or larger caches; for caches without subblock placement, the mean of the data cache penalty was often higher than 50%. We also found that a cache size of 512K allowed the allocation area for six of the benchmark programs to fit in the cache, which substantially improved the performance of cache organizations without subblock placement.

The implications of these results are clear. First, a stack is not needed to achieve good memory subsystem performance. Given the right memory subsystem, heap allocation of procedure activation records can also have good memory subsystem performance. Heap allocation can be used without a performance penalty in place of stack allocation, even though it is a more general storage management technique. Second, computer architects can better support modern languages which make heavy use of dynamic storage allocation on machines with small primary caches by using subblock placement with a subblock size of 1 word.

8 Acknowledgements

We would like to thank Edoardo Biagioni, Brad Chen, Olivier Danvy, Alessandro Forin, Urs Hoelzle, Kathryn McKinley, Erich Nahum, Darko Stefanović, and Daniel Stodolsky for comments on drafts of this paper. We thank Peter Lee for his encouragement and advice during this work. We thank Brian Milnes and the facilities at CMU for setting up the hardware according our every whim. We thank Tom Dewey for explaining the partial-word write mechanism in the DS5000/200 to us. We thank Andrew Appel, Dave MacQueen and many others for creating SML/NJ. We thank James Larus for creating qpt and for answering the questions which arose while we were extending his tool. We thank Mark Hill for creating his cache simulator Tycho. Last but not least, we thank all the members of the Fox project for their interest in this work and for accommodating our demand for compute cycles.

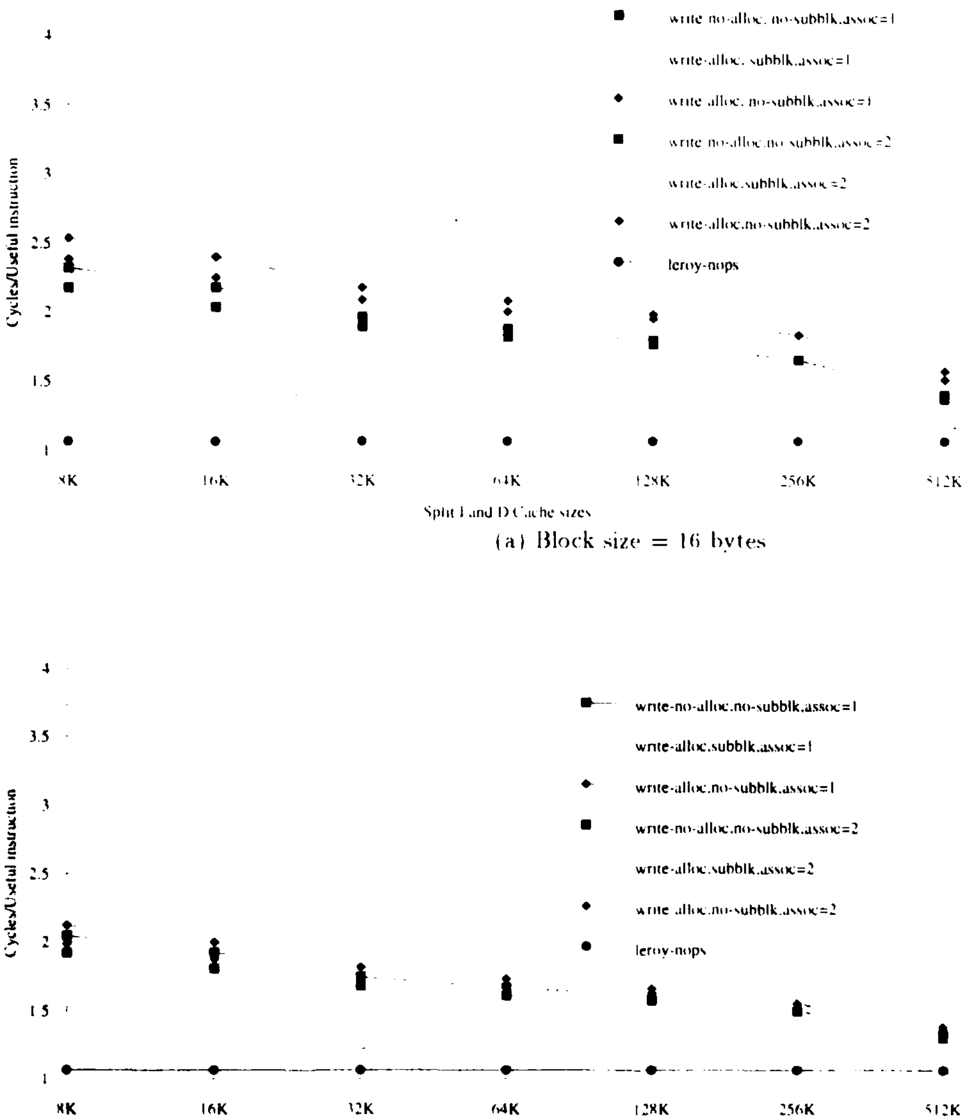
A Summary Graphs



(a) Block size = 16 bytes

(b) Block size = 32 bytes

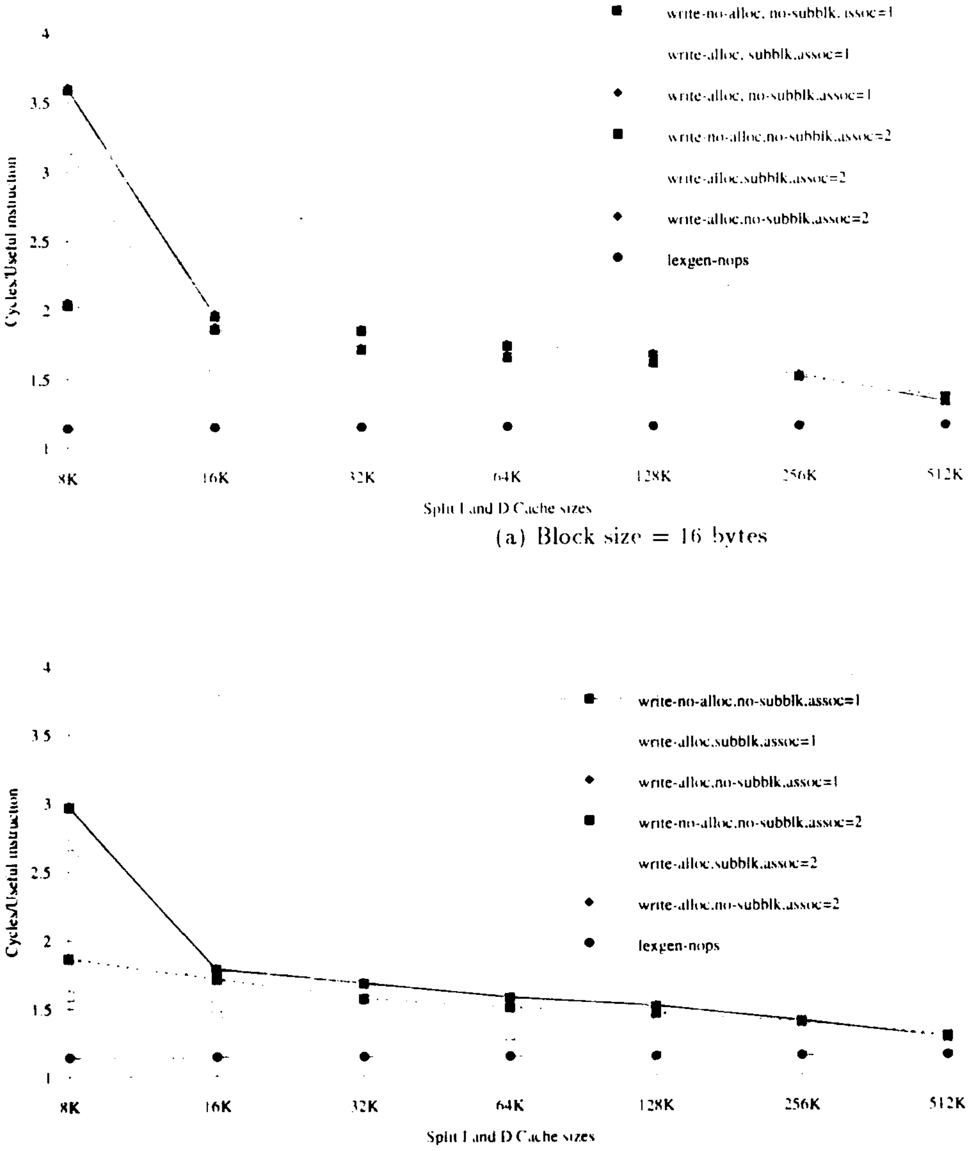
Figure 9: CW summary



(a) Block size = 16 bytes

(b) Block size = 32 bytes

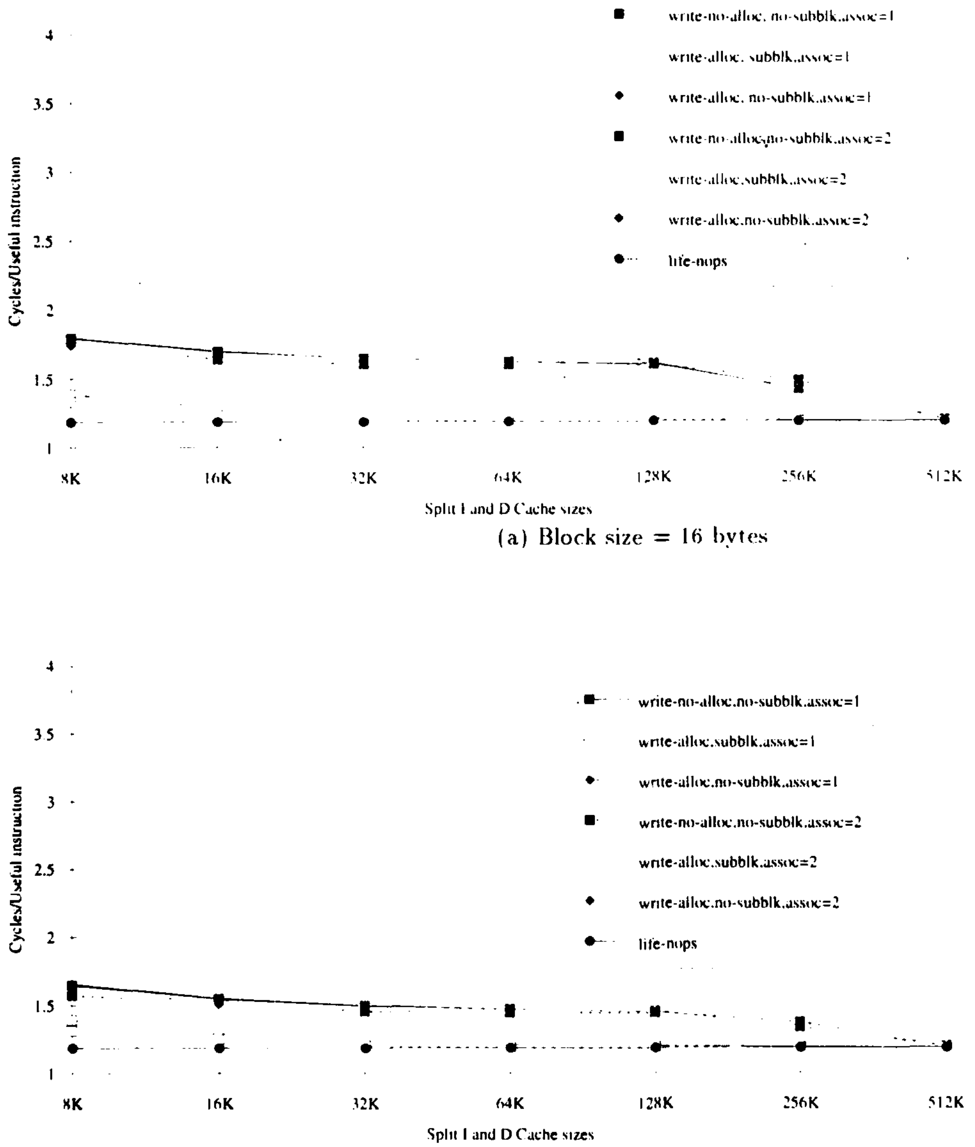
Figure 10: Knuth-Bendix summary



(a) Block size = 16 bytes

(b) Block size = 32 bytes

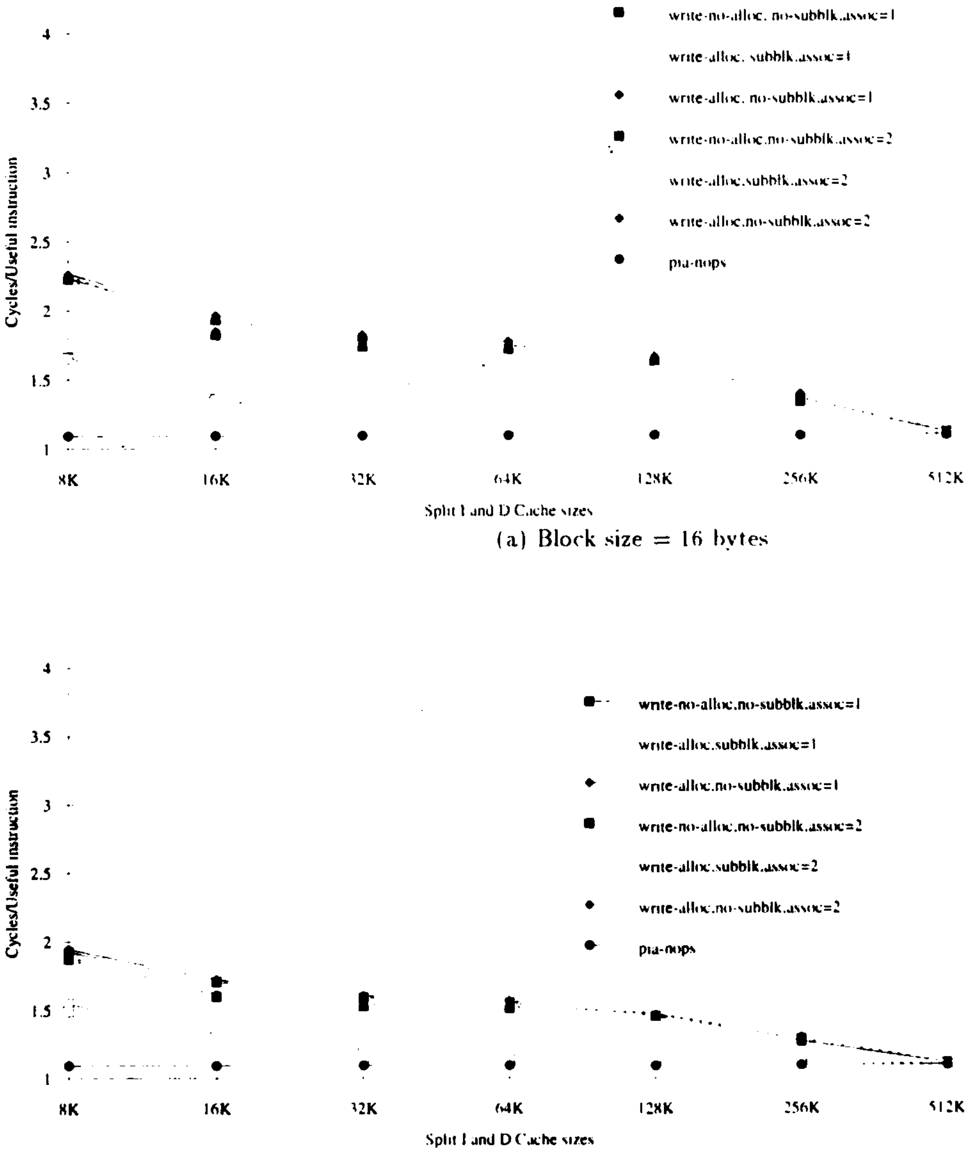
Figure 11: Lexgen summary



(a) Block size = 16 bytes

(b) Block size = 32 bytes

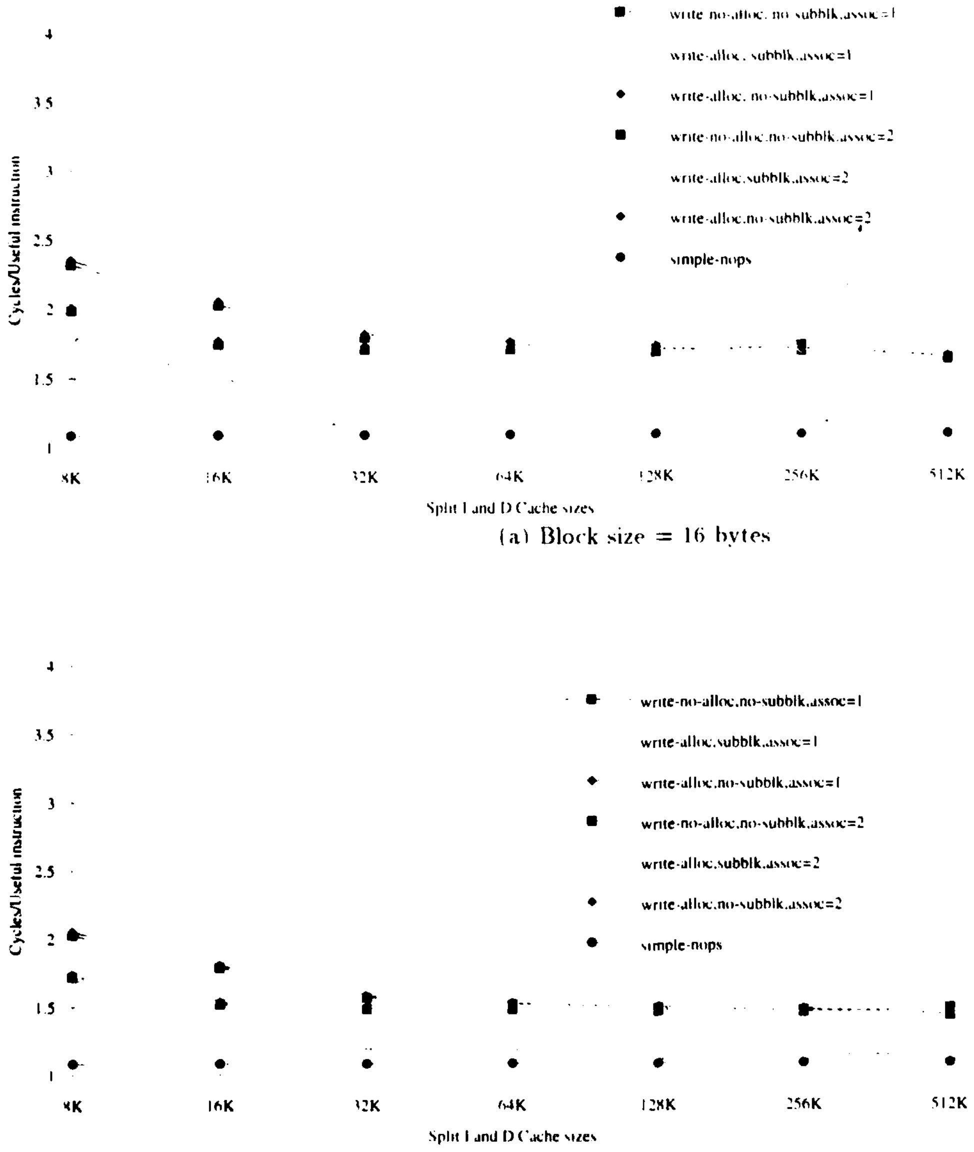
Figure 12: Life summary



(a) Block size = 16 bytes

(b) Block size = 32 bytes

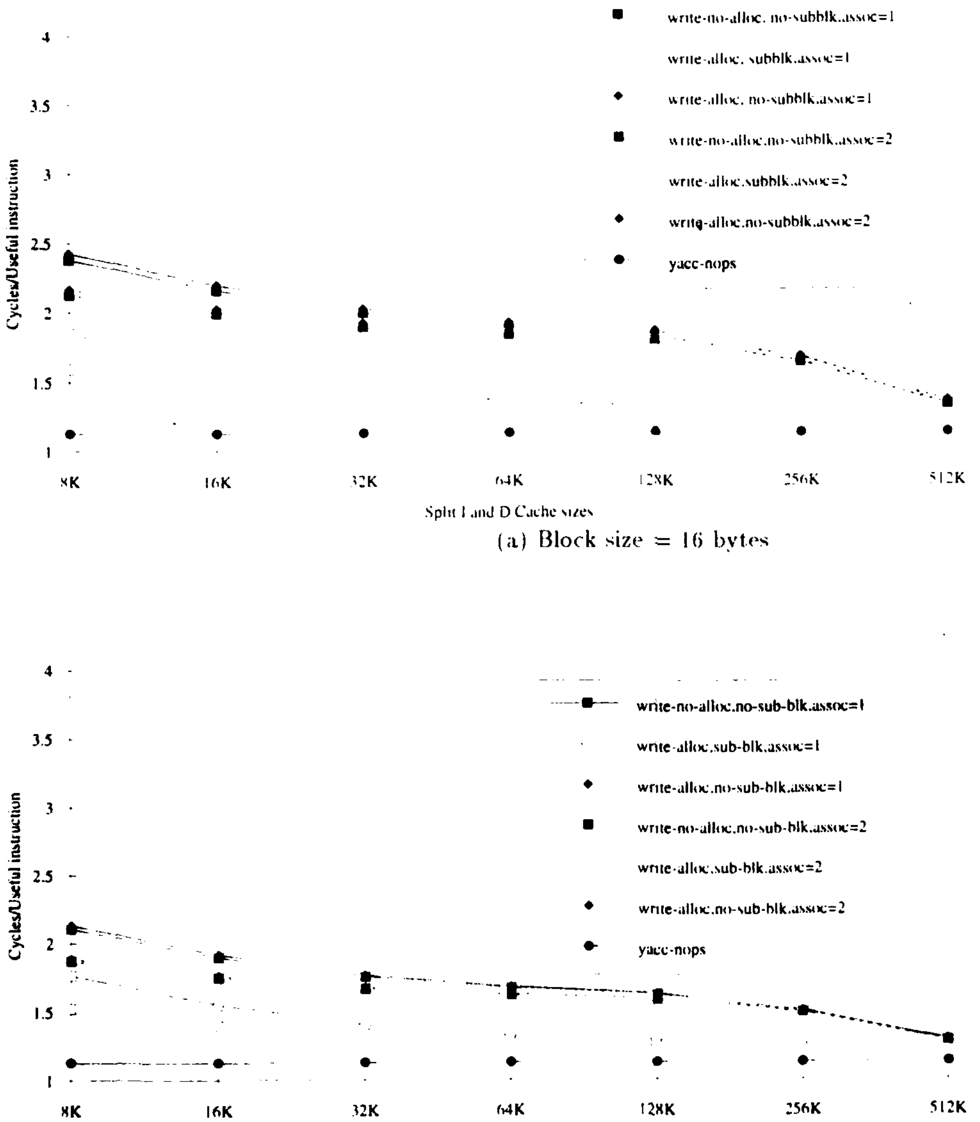
Figure 13: PIA summary



(a) Block size = 16 bytes

(b) Block size = 32 bytes

Figure 14: Simple summary



(a) Block size = 16 bytes

(b) Block size = 32 bytes

Figure 15: YACC summary

References

[1] Appel, A. W. Garbage collection can be faster than stack allocation. *Information Processing Letters 25*, 4 (1987), 275–279.

[2] Appel, A. W. Simple generational garbage collection and fast allocation. *Software – Practice and Experience 19*, 2 (Feb. 1989), 171–184.

[3] Appel, A. W. A Runtime System. *Lisp and Symbolic Computation 3*, 4 (Nov. 1990), 343–380.

[4] Appel, A. W. *Compiling with Continuations*. Cambridge University Press, 1992.

[5] Appel, A. W. Personal communication, March 22 1993.

[6] Appel, A. W., and Jim, T. Y. Continuation-Passing, Closure-Passing Style. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages* (Austin, Texas, Jan. 1989), ACM, pp. 293–302.

[7] Appel, A. W., Mattson, J. S., and Tarditi, D. A lexical analyzer generator for Standard ML. Distributed with Standard ML of New Jersey, 1989.

[8] Appel, A. W., and Shao, Z. Callee-save registers in continuation-passing style. *Lisp and Symbolic Computation* (1992), 191–221.

[9] Ball, T., and Larus, J. R. Optimally profiling and tracing programs. In *19th Symposium on Principles of Programming Languages* (Jan. 1992), ACM.

[10] Borg, A., Kessler, R. E., Lazana, G., and Wall, D. W. Long address traces from RISC machines: Generation and analysis. Tech. Rep. 89/14, DEC Western Research Laboratory, Sept. 1989.

[11] Case, B. PA-RISC provides rich instruction set within RISC framework. *Microprocessor Report 5*, 6 (April 1991).

[12] Chen, J. B., and Bershad, B. N. The impact of operating system structure on memory system performance. In *Fourteenth Symposium on Operating System Principles* (Dec. 1993), ACM.

[13] Chen, T.-F., and Baer, J.-L. Reducing memory latency via non-blocking and prefetching caches. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, Sept. 1992), ACM, pp. 51–61.

[14] Cheney, C. A nonrecursive list compacting algorithm. *Communications of the ACM 13*, 11 (Nov. 1970), 677–678.

[15] Cleaveland, R., Parrow, J., and Steffen, B. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *Transactions on Programming Languages and Systems 15*, 1 (Jan. 1993), 36–72.

[16] Crowley, W. P., Hendrickson, C. P., and Rudy, T. E. The SIMPLE code. Tech. Rep. UCID 17715, Lawrence Livermore Laboratory, Livermore, CA, Feb. 1978.

[17] Cypress Semiconductor, Ross Technology Subsidiary. *SPARC RISC User's Guide*, second ed., Feb. 1990.

[18] Digital Equipment Corporation. *DS5000/200 KN02 System Module Functional Specification*.

[19] Digital Equipment Corporation. *DECStation 3100 Desktop Workstation Function Specification*. 1.3 ed. Palo Alto, CA, August 1990.

[20] Digital Equipment Corporation. *DECchip 21064-AA Microprocessor Hardware Reference Manual*, first ed. Maynard, MA, Oct. 1992.

[21] Ekanadham, K., and Arvind. SIMPLE: An exercise in future scientific programming. Technical Report Computation Structures Group Memo 273, MIT, Cambridge, MA, July 1987. Simultaneously published as IBM/T. J. Watson Research Center Research Report 12686, Yorktown Heights, NY.

[22] Fenichel, R. R., and Yochelson, J. C. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM 12*, 11 (Nov. 1969), 611–612.

[23] Hieb, R., Dybvig, R. K., and Bruggeman, C. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation* (White Plains, New York, June 1990), ACM, pp. 66–77.

[24] Hill, M., and Smith, A. Evaluating associativity in CPU caches. *IEEE Transactions on Computers 38*, 12 (Dec. 1989), 1612–1630.

[25] Hill, M. D. A case for direct mapped caches. *Computer 21*, 12 (Dec. 1988), 25–40.

[26] Jouppi, N. P. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (Seattle, Washington, May 1990), pp. 364–373.

[27] Jouppi, N. P. Cache write policies and performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (San Diego, California, May 1993), pp. 191–201.

[28] Kane, G., and Heinrich, J. *MIPS RISC Architecture*. Prentice-Hall, 1992.

[29] Kessler, R. E., and Hill, M. D. Page placement algorithms for large real-indexed caches. *Transactions on Computer Systems 10*, 4 (Nov. 1992), 338–359.

[30] Koopman, Jr., P. J., Lee, P., and Siewiorek, D. P. Cache behavior of combinator graph reduction. *Transactions on Programming Languages and Systems 14*, 2 (Apr. 1992), 265–277.

[31] Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., and Adams, N. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN '86 Conference Symposium on Compiler Construction* (Palo Alto, California, June 1986), ACM, pp. 219–233.

[32] Larus, J. R. Abstract Execution: A technique for efficiently tracing programs. *Software Practice and Experience 20*, 12 (Dec. 1990), 1241–1258.

[33] Larus, J. R., and Ball, T. Rewriting executable files to measure program behavior. Tech. Rep. Wis 1083, Computer Sciences Department, University of Wisconsin-Madison, Mar. 1992.

[34] Mattson, R. L., Gecsei, J., Slutz, D. R., and Traiger, I. L. Evaluation techniques for storage hierarchies. *IBM Systems Journal 9*, 2 (1970), 78–117.

[35] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[36] Mogul, J. C., and Borg, A. The effect of context switches on cache performance. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, California, Apr. 1991), pp. 75–84.

[37] Patterson, D. A., and Hennessy, J. L. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, California, 1990.

[38] Peng, C.-J., and Sohi, G. S. Cache memory design considerations to support languages with dynamic heap allocation. Tech. Rep. 860, Computer Sciences Department, University of Wisconsin-Madison, July 1989.

[39] Przybylski, S. A. *Cache and Memory Hierarchy Design: A Performance-Directed Approach.* Morgan Kaufmann Publishers, San Mateo, California, 1990.

[40] Reade, C. *Elements of Functional Programming.* Addison-Wesley, Reading, Massachusetts, 1989.

[41] Rees, J., and Clinger, W. Revised report on the algorithmic language Scheme. *SIGPLAN Notices 21*, 12 (Dec. 1986), 37–79.

[42] Short, R. T., and Levy, H. M. A simulation study of two-level caches. In *15th Annual International Symposium on Computer Architecture* (June 1988), pp. 81–89.

[43] Slater, M. PA workstations set price/performance records. *Microprocessor Report 5*, 6 (Apr. 1991).

[44] Tarditi, D., and Appel, A. W. ML-YACC, version 2.0. Distributed with Standard ML of New Jersey, Apr. 1990.

[45] Tarditi, D., and Diwan, A. The full cost of a generational copying garbage collection implementation. OOPSLA '93 Workshop on Memory Management and Garbage Collection, Sept. 1993.

[46] Uhlig, R., Nagle, D., Mudge, T., and Sechrest, S. Software TLB management in OSF/1 and Mach 3.0. Tech. Rep. CSE-TR-156-93, University of Michigan, 1992.

[47] Waugh, K. G., McAndrew, P., and Michaelson, G. Parallel implementations from function prototypes: a case study. Tech. Rep. Computer Science 90/4, Heriot-Watt University, Edinburgh, Aug. 1990.

[48] Wilson, P. R., Lam, M. S., and Moher, T. G. Caching considerations for generational garbage collection: a case for large and set-associative caches. Tech. Rep. EECS-90-5, University of Illinios at Chicago, Dec. 1990.

[49] Wilson, P. R., Lam, M. S., and Moher, T. G. Caching considerations for generational garbage collection. In *1992 ACM Conference on Lisp and Functional Programming* (San Francisco, California, June 1992), pp. 32–42.

[50] Zorn, B. The effect of garbage collection on cache performance. Tech. Rep. CU-CS-528-91, University of Colorado at Boulder, May 1991.